AFRL-IF-RS-TR-2005-334
Final Technical Report
September 2005

ASBESTOS: SECURING UNTRUSTED SOFTWARE WITH INTERPOSITION

New York University

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. Q921

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

STINFO FINAL REPORT

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-2005-334 has been reviewed and is approved for publication

APPROVED: /s/

PATRICK M. HURLEY
Project Engineer

FOR THE DIRECTOR: /s/

WARREN H. DEBANY, JR., Technical Advisor
Information Grid Division
Information Directorate

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 074-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	SEPTEMBER 2005	Final Mar 04 – Mar 05

4. TITLE AND SUBTITLE
ASBESTOS: SECURING UNTRUSTED SOFTWARE WITH INTERPOSITION

5. FUNDING NUMBERS
C - FA8750-04-1-0090
PE - 62301E
PR - Q921
TA - GA
WU - 01

6. AUTHOR(S)
David Mazieres, Eddie Kohler,
Frans Kaashoek and Robert Morris

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
New York University
Department of Computer Science
715 Broadway, #708
New York New York 10003

8. PERFORMING ORGANIZATION REPORT NUMBER
N/A

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Defense Advanced Research Projects Agency
3701 North Fairfax Drive
Arlington Virginia 22203-1714

AFRL/IFGA
525 Brooks Road
Rome New York 13441-4505

10. SPONSORING / MONITORING AGENCY REPORT NUMBER
AFRL-IF-RS-TR-2005-334

11. SUPPLEMENTARY NOTES

AFRL Project Engineer: Patrick M. Hurley/IFGA/(315) 330-3624/ Patrick.Hurley@rl.af.mil

12a. DISTRIBUTION / AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

12b. DISTRIBUTION CODE

13. ABSTRACT ***(Maximum 200 Words)***

The main goal of the Asbestos effort was to build an operating system that allows users to control applications using encapsulation, without having to understand the application security properties. The specific tasks undertaken were to study interposition - as a mechanism for controlling software, to investigate extensions of the interface to mandatory access control, to work out detailed message sequences for example applications, and to develop a prototype implementation of Asbestos. In the end, after examination of example applications (a "hug-proof' web server) and our mandatory access control mechanism, led to the realization that the proper mandatory access control mechanism can suffice for the kinds of security properties we wished to achieve. Thus, the prototype implementation relies mostly on Asbestos's mandatory labeling mechanism for security, not interposition.

14. SUBJECT TERMS
Principles Of Least Privilege, Mandatory Access Control, Labeling For Access Control And Information Flow, Interposition

15. NUMBER OF PAGES
29

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Table of Contents

Summary

Asbestos, a new operating system we have prototyped, provides novel labeling mechanisms that makes it easy for programmers to develop secure software. Asbestos labels let applications be structured so as to tolerate flaws in major parts of the application without compromising security. Moreover, a new label save/restore mechanism pushes beyond the traditional process abstraction to avoid continually accumulating restrictions when manipulating data in different compartments. For example, a Web server that uses Asbestos labels to implement mandatory controls on client data requires only one virtual memory page per user.

Introduction

Today's software systems are insecure: floods of security advisories from vendors and security organizations document a steady stream of high-profile vulnerabilities in end-user applications, operating systems, and even routers, leading to widespread attacks, often estimated to incur billions of dollars in direct and indirect costs.

Many vulnerabilities stem from conflicts between the needs of application developers and the basic principles for building secure computer systems, such as giving applications minimal privilege. Violating the principles--by assuming elevated privilege, for example--makes development so much easier on conventional operating systems that it's doubtful the principles will ever be broadly followed there.

Because one cannot hope to fix or even understand all the software people need to run, securing systems boils down to the problem of reasoning about the behavior of large amounts of software without necessarily understanding the software itself, a task that might be accomplished by redefining the interface between software and the underlying operating system.

The goal of this project was to design a new operating system, called Asbestos, that would make it easy to control, understand, and observe interactions among applications without understanding the applications themselves. This approach enables people to monitor and control systems and enforce a wide range of security policies. A policy may be a set of static constraints enforced by the system or, more generally, a program that imposes a narrow interface for exporting information or accessing other parts of the system. Such a system could apply towards platforms ranging from resource-poor sensors to high-end servers, to help enforce meaningful security policies on new applications or privilege-hungry imported legacy code.

The scope of the project was to flesh out the design of Asbestos and to implement a skeleton prototype to explore key design decisions. The design work focused on

enforcing both discretionary and mandatory access control with Asbestos, and on exploring example applications. The detailed results are documented in Appendixes A and B.

This project was a short-term seedling effort, which we hoped would lead to a larger follow-on project for Asbestos. Indeed, the most important result is that this project helped lay the groundwork for the following major implementation effort, jointly funded by DARPA and an NSF Cybertrust grant (CNS-0430425).

Methods, Assumptions, and Procedures

Asbestos is organized around the idea of exposing and controlling messages. Every interaction between an application and the system or another application is represented as a message. The key problems the operating system faces are ensuring that messages encapsulate all interactions, understanding what parts of the system may be observed or modified with a given message, and most importantly making the mechanisms that control interaction available to unprivileged software, so that security policies need not all be imposed from highly privileged code.

The Asbestos design solves these problems using a novel label scheme that can be viewed as an extension of capabilities to provide decentralized Mandatory Access Control (MAC). Previously, capability-based systems have only achieved MAC by either marrying two very different security mechanisms (such as capabilities and a completely separate labeling system), or by granting processes completely disjoint sets of capabilities so as to achieve heavy-weight isolation.

Our labeling scheme furthermore has the advantage of being decentralized--so that even unprivileged processes can make use of the Operating System's (OS) mandatory access control primitives to control the flow of information. In traditional terms, unprivileged application can on-the-fly create compartments that protect the secrecy (or integrity) of data and processes they contain. The process that creates a compartment controls what information can leave the compartment. This ability--to downgrade information within a compartment--is equivalent to possessing a capability in a traditional discretionary capability system. The system can be used in a degenerate way, in which the ability to downgrade is simply delegated around and conveys the ability to invoke services within a compartment. However, applications developed for the capability model can have mandatory access control imposed by other compartments.

Another limitation of previous operating systems is that the granularity of compartments is too coarse. We designed a novel virtual memory system that can control the flow of information, even within a single process. We identified a target application of a possibly buggy web server handling sensitive information.

Results and Discussion

The result of this seedling is that it has grown into a full-fledged OS development project. The applications we investigated—in particular, protecting information in web servers--turn out to be both important and hard to address with existing operating systems. Moreover, the initial success of our follow-on DARPA/NSF project suggests the approach may be effective and practical.

More specifically we have prototyped both the Asbestos kernel, and a web server running on top of Asbestos. The web server is designed around special Asbestos virtual memory system, which allows labels to be applied at the granularity of individual pages, so that one can control the flow of information even within one process. Thus, even software bugs in the web server cannot cause one user to receive another's private data. The system requires only one or two pages of memory per active user--immensely less than traditional operating systems, which would require one process per compartment. The details are in Appendixes A and B.

Related work

A number of previous systems have individually provided either the ability to isolate untrusted software, labels, low-level interposition facilities, or consistent intelligible interfaces to different types of resource.

Message-based operating systems, such as Accent, Amoeba, Chorus, L4, Spring, and V can isolate system services by running them as independent, user-level processes, and provide natural support for interposition through message-based interfaces. However, none of these systems can provide the combined security and flexibility of Asbestos. For example, Amoeba bases access control on self-authenticating capabilities, precluding policies that restrict delegation. L4 uses a strict hierarchy of interpositions, useful for confining executable content, but not amenable to composition of independent restrictions on information flow imposed by mutually distrustful parties.

Conclusions

Asbestos is a new operating system with a labeling mechanism that promises to enforce security properties on applications without needing to trust the bulk of the software running on a system. Asbestos labels are in some ways similar to previous operating

systems with mandatory access control, but with several novel properties, including the ability for unprivileged software to create compartments on-the-fly, decentralized control over sanitization of data, and the ability to revert the state of a process, so as to let software safely messages bearing data from multiple compartments. We hope that these properties will allow Asbestos to apply mandatory access control to a wider range of problems than in previous systems, and furthermore avoid the notorious problem of "accumulating taints" on processes. The initial prototype shows promise, and the seedling has led into a full-fledged implementation effort co-sponsored by DARPA and NSF.

References

The following two references are included as appendixes:

Appendix A - Maxwell Krohn, Petros Efstathopoulos, Cliff Frey, Frans Kaashoek, Eddie Kohler, David MaziËres, Robert Morris, Michelle Osborne, Steve VanDeBogart, and David Ziegler. *Make Least Privilege a Right (Not a Privilege)*. In Proceedings of the 10th Workshop on Hot Topics in Operating Systems, Santa Fe, NM, June 2005.

Appendix B - Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazières, Frans Kaashoek, and Robert Morris. *Labeling virtual memory in the asbestos operating system.* In Proceedings of the 20th ACM Symposium on Operating Systems Principles, Brighton, UK, October 2005. *(The version as submitted is included in the appendix; the final version will not be done until August 2005)*

Appendix A - Make Least Privilege a Right (Not a Privilege)

Maxwell Krohn*, Petros Efstathopoulos†, Cliff Frey*, Frans Kaashoek*, Eddie Kohler†,
David Mazières‡, Robert Morris*, Michelle Osborne‡, Steve VanDeBogart† and David Ziegler*
*MIT †UCLA ‡NYU
asbestos@scs.cs.nyu.edu

ABSTRACT

Though system security would benefit if programmers routinely followed the *principle of least privilege* [24], the interfaces exposed by operating systems often stand in the way. We investigate why modern OSes thwart secure programming practices and propose solutions.

1 INTRODUCTION

Though many software developers simultaneously revere and ignore the principles of their craft, they reserve special sanctimony for the *principle of least privilege*, or *POLP* [24]. All programmers agree in theory: an application should have the minimal privilege needed to perform its task. At the very least, developers must follow five *POLP requirements*: (1) split applications into smaller protection domains, or "compartments"; (2) assign exactly the right privileges to each compartment; (3) engineer communication channels between the compartments; (4) ensure that, save for intended communication, the compartments remain isolated from one another; and (5) make it easy for themselves, and others, to perform a security audit.

Unfortunately, modern operating systems render the application of these requirements onerous, dangerous, or impossible. In our experience (detailed in Section 2.2), building least-privileged software is cumbersome and labor-intensive: following POLP feels more like an abuse of the operating system's interface than a judicious use of its features. Most programmers spare themselves these difficulties by reverting to monolithic, over-privileged application designs. Unsurprisingly, this exposes machines to attacks both old (remote attacks on privileged servers) and new ("install attacks", which take advantage of users' willingness to run high-privilege installers to infect machines with adware, spyware, or malware). We cannot write bug-free applications or prevent honest users from occasionally executing malicious code. Instead, our best hope is to contain the damage of evil code by resurrecting POLP.

In this paper, we examine some ways that current OSes discourage development of least-privilege applications (Section 2), then propose OS design ideas that might encourage it instead. A first approximation of a POLP-friendly system is one based on *capabilities*, discussed in Section 3. Though capabilities have historically flummoxed application designers, we present a more usable interface, based on the familiar Unix file system. In Section 4, we discuss shortcomings in this proposed design: weaknesses in the separated system might still allow vulnerabilities to spread, and process-sized compartments are too coarse-grained. We then propose a solution based on *decentralized mandatory access control* [17]. The end result is a new operating system called *Asbestos*.

2 LESSONS FROM CURRENT SYSTEMS

Modern Unix-like operating systems provide a limited API for running programs according to POLP. We examine how far administrators and programmers can push these features if POLP is their goal.

2.1 `chrooting` or `jailing` Greedy Applications

Because Unix grants privilege with coarse granularity, many Unix applications acquire more privileges than they require. These "greedy applications" can be tamed with the `chroot` or `jail` system calls. Both calls confine applications to *jails*, areas of the file system that administrators can configure to exclude setuid executables and sensitive files. FreeBSD's `jail` goes further, restricting a process's use of the network and interprocess communication (IPC). System administrators with enough patience and expertise can `chroot` or `jail` standard servers such as Apache [1], BIND [3] and sendmail [26], though the process resembles stuffing an elephant into a taxicab.

Even when possible, the `chroot` and `jail` approaches face more fundamental drawbacks:

Jails are heavyweight. The jailed file system must contain copies of system-wide configuration files (such as `resolv.conf`), shared libraries, the run-time linker, helper executable files, and so on. Maintaining collections of duplicated files is an administrative difficulty, especially on systems with many jailed applications.

Jails are coarse-grained. Running a process in a jail is similar to running it on its own virtual machine. Two jailed applications can share files only if one's namespace is a superset of the other, or if inefficient workarounds are used, such as NFS-mounting a local file system.

Jails require privilege. Unprivileged users may not call `chroot` or `jail`.[1] Jails are therefore ill-suited for containing the many untrusted applications that should not have privileges, such as executable email attachments or browser plugins.

Finally, `chroot` or `jail`'s *ex post facto* imposition of security is no substitute for POLP-based design. For example, a typical dynamic content Web server (such as Apache with PHP [18]) runs many logically unrelated scripts within the same address space. A vulnerability in

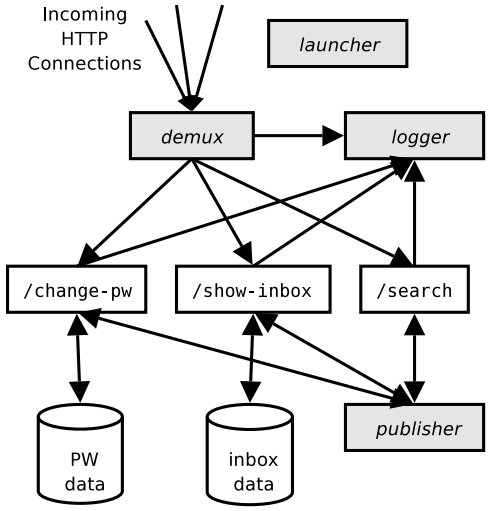

Figure 1: Block diagram of the OKWS system. Standard processes are shaded, while site-specific services and databases are shown in white. The privileged *launcher* process launches the *demux*, *publisher*, *logger* and the site-specific services. The databases shown might either be running locally, or on different machines.

any one script exposes all other scripts to attack, regardless of whether the server is jailed.

2.2 Ad-Hoc Privilege Separation

True privilege separation is possible on Unix through a collection of ad-hoc techniques. For instance, our POLP-based OK Web Server (OKWS) [12] uses a pool of worker processes to sequester each logical function (i.e. `/show-inbox`, `/change-pw`, and `/search`) of the site into its own address space. The *demux*, a small, unprivileged process, accepts incoming HTTP requests, analyzes their first lines, and forwards them to the appropriate workers using file descriptor passing. Workers then respond to clients directly. A privileged *launcher* process starts the suite of processes, ensuring that all are jailed into empty subtrees of the file system, and that they do not have the privileges to interact with one another. Finally, since workers' `chroot` environments prohibit them from accessing the root file system directly, they write HTTP log entries and read static HTML content via small, unprivileged helper processes: the *logger* and the *publisher*, respectively. Figure 1 shows a block diagram of a simple OKWS configuration.

The goal of this design is to separate application logic into disjoint compartments, so that any local vulnerability (especially in site-specific work processes) cannot spread. In particular, workers cannot send each other signals or trace each other's system calls, they cannot access each other's databases, no worker can alter any executable or library, and workers cannot access each other's coredumps. Unfortunately, achieving these natural requirements complicates OKWS. Its launcher must:

1. Establish a `chroot` environment, with the correct file system permissions, that contains the appropriate shared libraries, configuration files, run-time linker, and worker executables.
2. Obtain unused UID and GID ranges on the system.
3. Assign the *i*th worker its own UID u_i and GID g_i.
4. Allocate a writable coredump directory for each UID.
5. Change the *i*th worker's executable to have owner `root`, group g_i, and access mode `0410`.
6. Call `chroot`.
7. For each worker process *i*: kill all processes running as user u_i or group ID g_i; fork; change user ID to u_i and group ID to g_i; `chdir` into the dedicated dump directory; and call `exec` on the correct executable.

The `chown` call in Step 5, the `chroot` call in Step 6, and the `setuid` call in Step 7 all require privileged system access, so the launcher must run as root. Unix offers no guarantees of an atomic UID reservation (as required in Step 2) or race-free file system permission manipulations (as required throughout). Even ignoring these potential security problems, this design requires involved IPC to coordinate worker and helper processes.

Other systems use similar techniques to solve related problems. Examples include remote execution utilities such as OpenSSH [23] and REX [10], and mail transfer agents such as qmail [2] and postfix [21]. Considering these applications and others, a trend emerges: in each instance, the intricate mechanics of privilege separation are invented anew. To audit the exact security procedures of these applications, one must comb tens of thousands of lines of code, each time learning a new system. Even automated tools that separate privileged operations [5] require root access.

2.3 A User-Level POLP Library?

At first glance, a user-level POLP library might seem able to abstract the security-related specifics of applications like OKWS, qmail, and so on. One such example of this approach is found in the Polaris system for Windows XP [30], which applies POLP to virus-prone client applications like Web browsers and spreadsheets[2] via `chroot`-like containers. Such solutions have three drawbacks. First, they require privileged access to the system. Second, libraries must work around the lack of good OS support for sharing across containers: since jailed processes work with copies of files, synchronization schemes are required to reconcile copies after changes. (For example, Polaris email plug-ins run in a jail with a copy of the attachment; a persistent "synchronizer" process updates the original if the plug-in changes the copy.) Finally, we suspect that POLP techniques used in more complicated servers such as OKWS do not generalize well. As evidence, both OKWS and REX, an ssh-like login facility, use the same libraries (the SFS toolkit [16]) but share little security-related code. This comes as no surprise since the two have very different se-

curity aims: OKWS hides most of the file system, while REX exposes it to authorized users; OKWS must support millions of possible users, while REX serves only those with login access to a given machine; application designers can extend OKWS with site-specific code, while REX runs unmodified. Fitting both POLP usages into one general template seems a tall order.

2.4 Unix as a Capability System

One of the main difficulties with ad-hoc privilege separation is that starting with a privileged process and subtracting privileges is more cumbersome and error-prone than starting with a totally unprivileged process and adding privileges. Unix-like operating systems in general favor the subtractive model, while capability-based operating systems [4, 28] favor the additive one. But Unix file descriptors are in fact capabilities. By hobbling system calls sufficiently—either through system call interposition [7, 22] or small kernel modifications—we can emulate those semantics of capability-based operating systems that enable privilege separation.

The idea is to allow calls that use already-opened file descriptors (such as `read`, `write`, and `mmap`), but shut off all "sensitive" system calls, including those that create new capabilities (such as `open`), assign capabilities control of named resources (such as `bind`), and perform file system modifications, permissions changes, or IPC without capabilities (such as `chown`, `setuid`, or `ptrace`). In OKWS, the launcher could apply such a policy to the worker processes, which only require access to inherited or passed file descriptors. The launcher could run without privilege, and would no longer navigate the system call sequence seen in Section 2.2. By disabling all unneeded privileges, the operating system could enforce privilege separation by default.

This works because Unix's capability-like system calls are *virtualizable*. Processes are usually indifferent to whether a file descriptor is a regular file, a pipe to another process, or a TCP socket, since the same `read` and `write` calls work in all three cases. In practical terms, virtualization simplifies POLP-based application design. Splitting a system into multiple processes often involves substituting user-space helper applications for kernel services; for instance, OKWS services write log entries to the *logger* instead of a Unix file. With virtualizable system calls, user processes can mimic the kernel's interface; programmers need not rewrite applications when they choose to reassign the kernel's role to a process.

More important, virtualizable system calls enable *interposition*. If an untrustworthy process asks for a sensitive capability, a skeptical operator can babysit it by handing it a pipe to an interposer instead. The interposer allows harmless queries and rejects those that involve sensitive information. If the kernel API is virtualizable, then the operator need not even recompile the untrustworthy process to interpose on it.

Unfortunately, most Unix system calls resist virtualization. Some do not involve any capability-like objects; others use hard-wired capabilities hidden in the kernel, such as "current working directory" and "file system root". User-level emulation of these problematic calls—which include `open`—is messy, if not impossible; but scrapping `open` in the name of POLP seems unlikely to compel the average programmer.

3 Operating System Support for POLP

With the lessons from Unix, we can now imagine a POLP-friendly operating system interface, one in which all system calls are capability-based and virtualizable like `read` and `write`. Adding universal virtualization support to a Unix-like capability system would cover all five POLP requirements. With capabilities, application programmers can split their program into isolated compartments (#1 and #4), granting each compartment exactly the privileges necessary to complete its task (#2). With virtualization, programmers use standard interfaces and libraries for communication between these compartments (#3), and auditors can understand this communication by interposing at the interfaces (#5). A new take on capabilities—one whose Unix-like appearance would be friendlier to application programmers—could simplify the application of POLP. This section presents a hypothetical design for such a system, which we'll call *Asnix*.

3.1 Asnix Design

In Asnix, interactions between a process and other parts of the system take the form of *messages* sent to *devices*. Devices include processes and system services as well as hardware drivers. Messages follow the outline "perform operation *O* on capability *C*, and send any reply to capability *R*." The kernel forwards this message to the device that originally issued *C*. There are a small number of operation types, as in NFS [25] and Plan 9's 9P [19]: `LOOKUP`, `READ`, `WRITE`, and so forth. The message types and their associated syntax are conventions; the kernel only enforces or interprets those messages sent to kernel devices. Requests and replies are sent and received asynchronously.

This design aids virtualization. All of a process's interactions with the system—whether with the kernel or other user applications—take the same form, explicitly involve capabilities, and shun implicit state. Consider, for example, the Unix call `open("foo")`. This call in Asnix would translate to a message that a process *P* sends to the file server device *FS*:

$$P \rightarrow \langle C_{\text{CWD}}, \text{LOOKUP}, \texttt{"foo"}, C_P \rangle \rightarrow FS.$$

The first argument is a capability C_{CWD} that identifies *P*'s current working directory. The second is the command to perform, the third represents the arguments, and the fourth is the capability to which the file system should send its response. Since Asnix makes explicit the CWD state hidden in the Unix system call, either the file server or a user process masquerading as the file server can answer the message.

3.2 Naming and Managing Capabilities

When an Asnix process P_1 launches a child process P_2, it typically grants P_2 a number of capabilities, ranging from directories on the file system to opened network connections. How can P_2 then access these capabilities? Traditional capability systems such as EROS favor global, persistent naming, but persistence has proven cumbersome to kernel and application designers [27].

Instead, we advocate a per-process, Unix-style namespace. Under Asnix, P_1 makes capabilities available to P_2 as files in P_2's namespace. Suppose P_1's namespace contains a tree of files and directories under `/secret`, and P_1 wishes to grant P_2 access to files under `/secret/bob`. As in Plan 9 [20], P_1 can mount `/secret/bob` as the directory `/home` in P_2's namespace. Unlike in Plan 9, the state implicit in the per-process namespace is handled at user level, and the kernel only traffics in messages sent to capabilities. For example, when the process P_2 opens a file under `/home`, the user level libraries translate the directory `/home` to some capability *C*. The kernel sees a LOOKUP message on *C*.

3.3 OKWS Under Asnix

We now consider what OKWS might look like on Asnix. Similar to before, the application suite consists of a *launcher*, *demux* and worker processes. Under Asnix, the logger process simply enforces append-only access to a log file, and might be useful for many applications (much like `syslogd` on today's systems). No publisher process is needed.

The launcher starts each worker process with an empty namespace (and thus no capabilities), then augments their namespaces as follows:

- In the *logger*'s namespace, mounts a logfile on `/okws/log`.
- In the *demux*'s namespace, mounts TCP port 80 on `/okws/listen`. For each worker process *i*, makes a socket pair and connects one end to `/okws/worker/`*i*.
- In worker process *i*'s namespace, mounts the other end of the above socket pair to `/okws/listen`. Mounts a connection to the logger on `/okws/log`. Mounts a read-only capability to the root HTML directory on `/www`.
- In all namespaces, makes required shared libraries available under `/lib`.

The launcher then launches all processes as before.

Under Unix, the launcher had to carefully construct jails, physically copying over files and invoking custom helper applications like the publisher and logger to limit file system access. Asnix, by contrast, lets the launcher expose capabilities to child processes at arbitrary points in their namespaces. Each child receives a synthetic file system perfectly suited to its task.

Moreover, all capabilities available to the Asnix OKWS processes are virtualizable. Workers accept connections on `/okws/listen` regardless of whether they originate from the kernel's TCP stack or the *demux*. Similarly, logging might be to a raw file or through a logging process that enforces append-only behavior; worker processes are oblivious to the difference.

3.4 Discussion

So far, the proposed system features no individually novel ideas; rather, it finds a new point in the OS design space amenable to secure application construction. Similar effects might be possible with message-passing microkernels, or unwieldy system call interposition modules. But in Asnix, the security primitives are few and simple, for both the kernel and application developer. Although the interface exposed to applications feels like the familiar Unix namespace (with added flexibility for unprivileged, fine-grained jails), an application's system interactions are entirely defined by its capabilities, and Asnix behaves like a capability system for the purposes of security analysis.

4 FINE-GRAINED POLP WITH MAC

Though we believe Asnix is an improvement over the status quo, it still falls short of enabling the high-level, end-to-end security policies we seek. Applications in Asnix can only express security policies in terms of *processes*, but processes often access many different types of data on behalf of different users. A security policy based on processes alone can therefore conflate data flows that ought to be handled separately. For example, OKWS on Asenix achieves the policy that data from a `/change-pw` process cannot flow to a corrupted `/show-inbox` process; but the policy says nothing about whether user *U*'s data within `/show-inbox` can flow to user *V*, meaning an attacker who compromises `/show-inbox` might be able to read an arbitrary user's private e-mail.

Of course, a much better policy for OKWS would be that "only user *U* can access user *U*'s private data". We would like to separate users from one another, much as we separated services in Section 3. Though a user session involves many different processes (such as the *demux*, databases[3], and worker processes), a policy for separating users should be achievable with a small, simple, isolated block of trusted code, as opposed to hidden authorization checks scattered throughout the system. This section extends Asnix to a new system, *Asbestos*, whose kernel uses flexible mandatory access control primitives to enforce richer end-to-end security policies. We are currently designing and building Asbestos as a full operating system for x86 machines.

4.1 Complete Isolation

One possible approach to better isolation, which we call *complete isolation*, would be to prohibit server-side pro-

cesses from speaking for multiple users. The server must be prepared to run a process for every service–user pair; trusted code in *demux* would route traffic accordingly. Similarly, a database process exists for each user, writing to a user-specific database file. Capabilities can guarantee separation between processes as usual. More drastic separation is possible with virtual machines [11, 32] so that each machine can only speak for one user.

Complete isolation hides a user's data from other users, but at significant cost. First, such systems are not scalable, requiring either an expensive fork-accept-close model or a huge pool of largely-idle per-user servers. Second, these systems do not accommodate convenient data sharing, even with trusted processes. While traditional systems could use simple SQL statements to aggregate statistics over rows of a site's databases, completely isolated systems would have to search millions of separate files, perhaps over NFS in the case of separated virtual machines. Separation in this case requires a tremendous sacrifice in flexibility for data management. Data will not flow where it shouldn't, because it cannot flow at all.

4.2 Decentralized, Fine-Grained MAC

Asbestos uses decentralized, fine-grained mandatory access control (MAC) primitives to solve this problem in a flexible and scalable manner. Subjects on the system, such as processes, I/O channels, and files, are assigned *labels*, and special privilege is needed to remove a label once applied. Furthermore, a subject transmits its labels to any other objects that it successfully communicates with. With labels, Asbestos tracks all subjects that have accessed a given object, whether directly or via proxy.

We propose two important modifications to traditional MAC-based operating systems. First, decentralization [17]: processes can create their own labeling schemes on the fly, so that a Web server can associate each remote user with her own label. Second, labels apply at the fine-grained level of individual memory pages, so that a single process can act on behalf of mutually distrustful users without fear of leaking data among them. Taken together, these two modifications allow application designers to dynamically partition server processes into isolated *sub-processes*, where a sub-process consists of a set of virtual pages that share the same label.

When a server process receives a message, it is automatically assigned to a sub-process based on the label of the message's source. Processing a message from user *U* "contaminates" the process with *U*'s labels. As in traditional MAC, contamination with the label *U* prevents a process from accessing resources forbidden from user *U*, such as user *V*'s network connection. Thus, the kernel must allow a process speaking on behalf of multiple users to purge its labels without leaking data. Asbestos lets a process flush its register state, remap its memory, and clear its labels, allowing it to serve a request on behalf of a different user *V*. However, the system still accommodates trusted *declassifiers*, such as statistics collectors, that can act on behalf of multiple users and traverse sub-process boundaries within a virtual address space.

With decentralized, fine-grained MAC, OKWS can achieve a strong end-to-end security policy. The only trusted code is a *labeler* module upstream of *demux*, which works as follows. When user *U* connects to the Web server, the *labeler* peeks at the incoming TCP connection *T* and authorizes it based on session state or login information. If authorization succeeds, the *labeler* labels *T* with *U*'s label. Now, any process that reads from *T* and writes to memory will automatically tag that memory page with *U*'s label, and will therefore push that page into *U*'s sub-process. The kernel allows an unprivileged process to accumulate labels for different users (such as for *U* and *V*), but it forbids that process from writing to a network channel not labeled with both. Thus, if *U* compromises a server process and convinces it to read from *V*'s memory, the server process will acquire labels for both *U* and *V*, and therefore cannot write out to *T*.

4.3 Discussion

This decentralized MAC design, combined with the capability architecture from Section 3, makes POLP convenient and practical for an OKWS-like Web server. We have no proof that other applications would similarly benefit from Asbestos, but we are optimistic. Asbestos provides simple, flexible, and fine-grained mechanisms for achieving the five important POLP requirements without sacrificing performance.

5 Related Work

Asbestos proposes the marriage of previous ideas in systems: the capability-based operating system [4, 13, 28, 33], the per-process name space [20], the virtualizable kernel interface (the logical extension of system-call interposition libraries [7, 22]), and decentralized MAC [17].

Naturally, other operating systems predating Asbestos meet related design goals or offer similar features. Message-based operating systems such as L4, Amoeba, V, Chorus and Spring can isolate system services by running them as independent, user-level processes and provide natural support for interposition through message-based interfaces [14]; Trusted Mach in particular views message-passing from a security perspective [6]. But ports in microkernel systems are coarse as capabilities go; for instance, a process can have a capability for the file server but not for a particular directory. For POLP, application programmers need arbitrary collections of specific capabilities; in this respect, the microkernels of yesteryear do not fit the bill.

The Flask System applies MAC to the Fluke Microkernel [29]. Many of Flask's core design principles have found a modern incarnation in SELinux [15], which, like TrustedBSD [31], adds mandatory access control to popular Unix systems. In both, static policy files dic-

tate which resources applications might access, and how processes can interact with one another. Such systems are attractive because they preserve the POSIX interface to which many programmers are accustomed. However, their policy extension model, which is based on privileged files and kernel modules, appears to fall short of the decentralized and uniformly-analyzable policies implemented by Asbestos labels.

Type safety is another way to enforce operating system security. Coyotos combines capabilities with language-level verification techniques [27]. Singularity combines strong isolation with a type-safe ABI [8]. At user level, the Java Sandbox uses customizable policies to specify an applet's access rights; dynamic sandboxing shows these policies can be automatically produced [9].

ACKNOWLEDGMENTS

The authors thank Lee Badger, Butler Lampson, Mike Walfish and the reviewers. This work was supported by DARPA grants MDA972-03-P-0015 and FA8750-04-1-0090, and by joint NSF Cybertrust/DARPA grant CNS-0430425. David Mazières and Robert Morris are supported by Sloan fellowships.

REFERENCES

[1] The Apache Software Foundation. Apache. `http://www.apache.org`.

[2] D. J. Bernstein. qmail. `http://cr.yp.to/qmail.html`.

[3] Internet Systems Consortium. Berkeley Internet Name Daemon. `http://www.isc.org/sw/bind`.

[4] A. C. Bomberger, W. S. Frantz, A. C. Hardy, N. Hardy, C. R. Landau, and J. S. Shapiro. The KeyKOS nanokernel architecture. In *USENIX Workshop on Microkernels and Other Kernel Architectures*. USENIX, 1992.

[5] D. Brumley and D. X. Song. Privtrans: Automatically partitioning programs for privilege separation. In *USENIX Security Symposium*, pages 57–72. USENIX, 2004.

[6] T. Fine and S. E. Minear. Assuring distributed trusted mach. In *Proceedings of the 1993 IEEE Symposium on Security and Privacy*, page 206, Washington, DC, USA, 1993. IEEE Computer Society.

[7] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. A secure environment for untrusted helper applications. In *Proceedings of the 6th Usenix Security Symposium*, San Jose, CA, USA, 1996.

[8] G. C. Hunt and J. R. Larus. Singulairty design motivation. Technical Report MSR-TR-2004-105, Microsoft Corporation, Dec. 2004.

[9] H. Inoue and S. Forrest. Anomaly intrusion detection in dynamic execution environments. In *NSPW '02: Proceedings of the 2002 workshop on New security paradigms*, pages 52–60. ACM Press, 2002.

[10] M. Kaminsky, E. Peterson, D. B. Giffin, K. Fu, D. Mazières, and M. F. Kaashoek. REX: Secure, extensible remote execution. In *Proceedings of the 2004 USENIX*, pages 199–212, Boston, MA, June–July 2004. USENIX.

[11] P. Karger, M. Zurko, D. Bonin, A. Mason, and C. Kahn. A retrospective on the VAX VMM security kernel. *Transactions on Software Engineering*, 17(11):1147–1165, 1991.

[12] M. Krohn. Building secure high-performance web services with OKWS. In *Proceedings of the 2004 USENIX*, Boston, MA, June–July 2004. USENIX.

[13] H. Levy. *Capability-based Computer Systems*. Digital Press, 1984.

[14] J. Liedtke. Toward real microkernels. *Communications of the ACM*, 39(9):70–77, 1996.

[15] P. Loscocco and S. Smalley. Meeting critical security objectives with security-enhanced linux. In *Proceedings of Ottawa Linux Symposium 2001*, June 2001.

[16] D. Mazières. A toolkit for user-level file systems. In *Proceedings of the 2001 USENIX*, pages 261–274. USENIX, June 2001.

[17] A. C. Myers and B. Liskov. A decentralized model for information flow control. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 129–142, Saint-Malo, France, October 1997. ACM.

[18] PHP: Hypertext processor. `http://www.php.net`.

[19] R. Pike, D. Presotto, S. Dorward, B. Flandrena, K. Thompson, H. Trickey, and P. Winterbottom. Plan 9 from Bell Labs. *Computing Systems*, 8(3):221–254, Summer 1995.

[20] R. Pike, D. Presotto, K. Thompson, H. Trickey, and P. Winterbottom. The use of name spaces in Plan 9. In *Proceedings of the 5th ACM SIGOPS Workshop*, Mont Saint-Michel, 1992.

[21] Postfix. `http://www.postfix.org`.

[22] N. Provos. Improving host security with system call policies. In *Proceedings of the 12th USENIX Security Symposium*, pages 257–271, Washington, DC, August 2003.

[23] N. Provos, M. Friedl, and P. Honeyman. Preventing privilege escalation. In *12th USENIX Security Symposium*, Washington, D.C., August 2003.

[24] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, Sept. 1975.

[25] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the Sun network filesystem. In *Proceedings of the Summer 1985 USENIX*, pages 119–130, Portland, OR, 1985. USENIX.

[26] The Sendmail Consortium. Sendmail. `http://www.sendmail.org`.

[27] J. S. Shapiro, M. S. Doerrie, E. Northup, S. Sridhar, and M. Miller. Towards a verified, general-purpose operating system kernel. In G. Klein, editor, *Proc. NICTA Formal Methods Workshop on Operating Systems Verification*, Sydney, Australia, 2004. NICTA Technical Report 0401005T-1, National ICT Australia.

[28] J. S. Shapiro, J. Smith, and D. J. Farber. EROS: a fast capability system. In *Proc. Symposium on Operating Systems Principles*, pages 170–185, 1999.

[29] R. Spencer, S. Smalley, P. Loscocco, M. Hibler, D. Andersen, and J. Lepreau. The flask security architecture: System support for diverse security policies. In *In Proceedings of the Eighth USENIX Security Symposium*, August 1999.

[30] M. Stiegler, A. H. Karp, K.-P. Yee, and M. Miller. Polaris: Virus safe computing for windows XP. Technical Report HPL-2004-221, December 2004.

[31] R. N. M. Watson. TrustedBSD: Adding trusted operating system features to FreeBSD. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, pages 15–28. USENIX Association, 2001.

[32] A. Whitaker, M. Shaw, and S. D. Gribble. Scale and performance in the Denali isolation kernel. In *Proceedings of the 2002 Symposium on Operating Systems Design and Implementation (OSDI)*, Dec. 2002.

[33] M. V. Wilkes and R. M. Needham. *The Cambidge CAP Computer and its Operating System*. North Holland, 1979.

NOTES

[1] Were it not for this prohibition, unprivileged users could use control of the `chroot`ed top-level directory to elevate privileges. The attack is to make a new directory `/tmp/foo`, hard link from `/tmp/foo/su` to the system `su`, write a new password file `/tmp/foo/etc/passwd`, call `chroot` on `/tmp/foo`, and then call `su` from within the jail.

[2] Polaris appears not as well-suited for larger servers.

[3] We assume for simplicity that databases run locally, though all concepts discussed can generalize to distributed deployments.

Appendix B
Labeling Virtual Memory in the Asbestos Operating System

Petros Efstathopoulos* Maxwell Krohn‡ Steve VanDeBogart*
Cliff Frey‡ David Ziegler‡
Eddie Kohler* David Mazières† Frans Kaashoek‡ Robert Morris‡
*UCLA ‡MIT †NYU
{pefstath,kohler,vandebo}@cs.ucla.edu, {krohn,frey,dziegler,kaashoek,rtm}@mit.edu, dm@nyu.edu

ABSTRACT

Widely-used operating systems do not provide adequate mechanisms for security-minded programmers: developing secure applications is complicated and error prone. Asbestos, a new operating system, provides novel labeling mechanisms that makes it easy for programmers to develop secure software. Asbestos labels let application developers create a wide range of easily understood access policies, both mandatory and discretionary; new label save/restore support makes it possible to implement efficient services with these labels. A Web server that uses Asbestos labels to implement mandatory controls on client data requires only one virtual memory page per user, demonstrating that the additional security comes at an acceptable cost.

1 INTRODUCTION

Today's computer systems have an undeniably bad track record in security. We routinely hear of Web servers [19] and other systems [28] experiencing catastrophic breaches that divulge tens or hundreds of thousands of people's private information. End users suffer from viruses and spyware that, through various applications, infiltrate their operating systems, leak clickstream data, send spam, participate in denial of service attacks, and perform other malicious actions.

Most of these problems can be attributed to two factors: exploitable flaws in software, and users' willingness to run malicious code disguised as legitimate software or documents. Unfortunately, neither factor appears likely to improve significantly in the near future. Thus, the most viable means of improving security in practice may be designing systems that accommodate these threats. For example, an email reader should be able to confine an executable attachment by only giving it access to a display window and perhaps a temporary file system. A Web site should be able to ensure that one user's private data cannot be sent to another user's browser by a buggy Web server.

Confining processes and limiting information flow require the ability to enforce nondiscretionary security policies. To date, most operating system support for nondiscretionary policies has taken the form of multi-level secure (MLS) systems suitable for military-style classification policies [6]. MLS systems primarily allow security administrators to impose external constraints on existing software. The nondiscretionary access control mechanisms in these systems typically cannot be used by ordinary users to craft their own policies, cannot help developers restructure applications to tolerate breaches, and cannot be applied at a fine granularity to protect large numbers of users' data, as would be required for a typical Web site. Language-based information flow systems can support somewhat more decentralized policies [26], but there are significant advantages to implementing flow control at the OS level—not least the smaller trusted computing base and hardware support for protection. Furthermore, even these systems may not support the dynamic, decentralized creation of principals. On the other hand, capability-based operating systems offer some attractive features, including dynamic principal creation and fine-grained access control but give up explicit control of information flow.

Motivated by the difficulty of writing secure code for today's operating systems, and believing that a clean-slate design would more likely lead to advances that could eventually be mapped back to more conventional OSes, we describe here a new operating system, Asbestos, that combines the advantages of capability-based and nondiscretionary-access systems. Asbestos access control is based on a single simple primitive, *Asbestos labels*, that can implement both discretionary and nondiscretionary access policies, in a completely decentralized fashion. Any process may create an access control space, represented by a *handle*, and control the policies applied relative to that space. Labels straightforwardly implement traditional capabilities, mandatory access control, and hybrid schemes.

But process-granularity labels themselves are not sufficient to build fast, secure applications. A Web service, for example, may speak concurrently to multiple users. The service author might like to enforce a policy in which no user could see or touch another user's data. Process labels would force such a service to be implemented with one process per user, at significant resource cost. Asbestos therefore supports finer-granularity access control, called *label save/restore*, that lets a service apply labels selectively to specific memory pages. The operating system ensures, with label checks and virtual memory operations, that pages for a given user are visible only during the processing of requests for that user. Even if one user's instance of the process is broken into, all other user data is safe. This primitive not only helps make applications more secure, it also might facilitate new types of ser-

vices: users could safely be granted more control over the code running on their behalf, since labels ensure that other users' data will never be compromised. Our measurements indicate that label save/restore lets an operating system run a Web server with the same, or more comprehensive, access control checks as a one-service-per-user model, but while using just one page of memory per user. Our implementation runs on real x86 hardware. Despite a completely untuned implementation, performance implications of label save/restore are modest.

The contributions of this paper include Asbestos labels, label save/restore, and our example application, a dynamic Web server using labels and label save/restore to provide memory efficient, safe support for multiple-user services.

1.1 A motivating application: a better web server

Dynamic Web servers often serve as gateways to databases containing private information. In this role, a Web server is expected to provide users with the data they require without exposing data that should remain private. Unfortunately, as Web sites grow in size and complexity, Web developers are more likely to forget or misapply access controls. Worse still, access-control techniques applied at the application level are vulnerable to attacks on an often bloated trusted computing base, including the operating system (e.g., Linux), the system library (e.g., `libc`), the Web server (e.g., Apache), any Web server modules (e.g., PHP or SSL), and the Web application itself (e.g., `change-pw.php`). A vulnerability at any level may allow an attacker to gain access to private data.

We believe three principles are necessary when building secure applications such as Web servers. First, whichever security primitives are invoked, they should be implemented with the smallest possible trusted computing base. Second, they should be mandatory as opposed to discretionary. Third, failures should be isolated; a bug in one function should not compromise all data.

Software such as the OK Web server (OKWS) has shown it is possible to contort the existing Unix interface to achieve some security goals, such as isolation, while ignoring the others [17]. In OKWS, each logical Web service (such as `change-pw` or `check-inbox`) runs as a separate process with its own address space. If an attacker compromises and controls `check-inbox`, he cannot collect or change arbitrary user passwords (e.g., `change-pw`).

With the benefit of Asbestos labeling, OKWS could do much better. On current operating systems, if a remote user *A* compromises an OKWS service (e.g. `check-inbox`), he would have all privileges of that service (e.g. read other users' email). Under Asbestos, even if a remote user *A* can compromise a service, kernel protections prevent him from accessing user *B*'s information. An OKWS-like Web server in Asbestos can achieve all three security principles.

2 Related Work

Labels have long been used to enforce mandatory access control and are required by higher divisions of the DoD Trusted Computer System Evaluation Criteria [6]. Security enhancement packages with labels are available today for popular operating systems such as Linux [21] and FreeBSD [36]. The idea of dynamically adjusting labels to track potential information flow dates back to the High-Water-Mark security model [18] of the ADEPT-50 in the late 1960s. Numerous systems have incorporated such mechanisms, including IX [23] and LOMAC [8].

Asbestos labels differ significantly from those of previous operating systems in several ways. In Asbestos, any process can dynamically create a label category, a handle, and control the propagation of information labeled with that category. Ordinary processes can both declassify information and raise the security clearance of other processes, but only in the particular categories they control. By contrast, traditional MAC systems have a fixed number of compartments and security levels, all under the control of the security administrator. The ORAC model [22] does support the idea of individual originators placing accumulating restrictions on data, but data can still only be sanitized by users with the privileged Downgrader role. Asbestos also differs from previous systems in that its virtual memory system allows labels to control the flow of information within a single process.

Asbestos labels more closely resemble language-level flow control mechanisms. Jif [27], in particular, was an inspiration for Asbestos because of its support for decentralized declassification through separate ownership of different label components. Because it is a programming language, Jif has the advantage of being able to perform most of its label checks statically, at compile time. Run-time checks can affect control flow on failure, thereby creating implicit information flows [5]. However, compared to Asbestos, Jif requires a centralized principal hierarchy and has no equivalent to the asymmetric label defaults Asbestos uses to support policies such as preventing one process from talking to another.

Because Asbestos handles are also communication endpoints, they can be thought of as capabilities. Many believe that capabilities cannot implement mandatory access control [2]. Strictly speaking, this is untrue. For instance, KeyKOS [15] achieved military-grade security by isolating processes into compartments and ensuring any capability pointing outside a compartment designated a special multilevel secure object. EROS [32] later successfully realized the principles behind KeyKOS on modern hardware.

Psychologically, however, people have not accepted pure capability-based confinement [24], perhaps from the fear that if just one inappropriate capability escaped, the whole system might collapse. As a result, a number of designs have combined capabilities with authority checks [1], interposition [12], or even labels [13]. Asbestos demonstrates that with

decentralized labels, capabilities are unnecessary—the labels themselves can be used to implement a capability system.

Asbestos handles, as communication end points, are also reminiscent of message-based operating systems [4, 20, 25, 30, 31, 33], some of which can confine executable content [11], others of which have had full-fledged mandatory access control implementations [3].

Mandatory access control can also be achieved with unmodified traditional operating systems through virtual machines [9, 14]. For example, the NetTop project [34] uses VMware for multi-level security. Virtual machines have two principal limitations, however: performance [16, 38] and coarse granularity. One of the goals of Asbestos is to allow very fine-grained information flow control, so that a single process can handle differently labeled data. To implement a similar structure with virtual machines would require a separate instance of the operating system for each label.

3 Operating System Primitives

Asbestos is a message-passing operating system. User-level processes communicate with one another via messages, which are sent to communication ports called *handles*. This section describes characteristics of these primitives important for understanding Asbestos's more unusual features.

3.1 Handles

Asbestos handles combine aspects of communication endpoints, capabilities, and kernel-protected information labels. An Asbestos handle is simply a 61-bit number, and applications can treat arbitrary numbers as handles. Two kernel structures monitor and control handle usage. A single *routing table* stores the *device* entity responsible for messages sent to each handle; and per-process *labels*, described further below, control information flow relative to individual handles. Different label settings can cause a handle to act like a capability, a multi-level security (MLS) level, or a combination of the two.

In this code fragment, an application creates a new handle. (Some arguments have been left out for conciseness.)

```
handle_t h;
r = sys_new_handle(&h, ...);
```

The **new_handle** system call generates a new handle, grants the process labels access to that handle (see below), stores the handle in the routing table, and optionally sets up an in-kernel message queue to receive messages sent to that handle. Each call to the **new_handle** system call returns a handle not seen before, until the space wraps around; at a rate of 1 billion handle creations per second, this would take about 73 years. **new_handle** results are ephemeral rather than persistent; a different handle sequence might arise from every boot. They are also unpredictable. There is no way to create a message queue for a known handle value, and the system encrypts new handle values with a block cipher. This closes certain covert channels—applications cannot tell how many other handles have been created—and will facilitate virtualization, since there are no known global handle values in the system.

Processes may try to send messages to any handle. The kernel checks the routing table to see if the handle has a controlling device (either a process message queue or some in-kernel entity). If so, the message is delivered there, subject to access control checks; otherwise, it is treated as a label check failure. Access control checks are implemented using labels, as described in section 4.

Other handle system calls include **handle_transfer**, by which one process can transfer ownership of a handle to another, and **set_handle_label**, which is described below. Each active handle corresponds to a 64-byte kernel-private data structure, called the *vnode*. The routing table simply maps handles to vnodes. Vnodes are reference counted; when all kernel references to a vnode disappear, the kernel may reuse its memory. However, the kernel will not reuse the handle associated with the vnode, since that handle might still be in use within a process.

3.2 Messages

Asbestos defines a small number of message types and conventions guiding their use. For example, the `LOOKUP` message type is used to look up entries in a directory or directory-like object. We intend all applications and OS services to use this small set of message types in a uniform manner, which simplifies virtualization of system functionality.

Messages have six components: a *destination handle*, a *type*, a *message code*, an *ID*, an optional *reply handle*, and an optional *payload*. The type defines the class of operation being requested. The device that receives the message will send replies to the reply handle. The message code can supply an argument for request types; for reply types, it reports any error result. Finally, the message ID helps match replies to the corresponding requests.

Message types include:

`LOOKUP` Looks up entries in directories or directory-like objects. The payload is the name of the entry to look up. Replies use type `LOOKUP_R` (a general convention); their payload generally contains one or more handle values for the entries.

`READ`, `WRITE` Requests data from an object, or writes data to an object.

`CONTROL` A catchall message for non-read/write access. The message code specifies behavior further (e.g. the file system responds to `STAT` control messages).

Messages are stored for delivery on in-kernel message queues. A message queue is associated with one or more handles, all of which must be controlled by the same process; all

messages sent to those handles are delivered to the message queue. Message queues are implemented as circular buffers of pages. Small message data is copied into the page containing the message header; large data uses copy-on-write page mappings. When delivering messages from the queue to the process, we map pages whenever possible. Message access control checks, which use labels, are performed at send time, but any changes to the receiving process's labels are delayed until the message is actually delivered.

3.3 System calls

Most Asbestos functionality, aside from sending and receiving messages and managing handles, will eventually be accessible through messages. Currently, though, we do support other system calls. Most importantly, Asbestos processes, like exokernel processes [7], can manage their virtual address space, and that of other processes they control, by mapping and unmapping pages to and from virtual addresses. The relevant system calls are **page_alloc**, **page_map** and **page_unmap**. However, Asbestos will not support writable shared memory: if two processes share a page, it is mapped either read-only or copy-on-write.

4 ASBESTOS LABELS

All Asbestos access control and information flow checks are implemented with a single primitive, *labels*. Labels are flexible enough to implement a wide range of discretionary and mandatory access policies, and the Asbestos label design is one of our main contributions. This section incrementally develops and explains that design. Labels have been used previously in operating systems and secure languages. Asbestos labels differ in their support for effective labels, temporary restrictions that can help implement discretionary policies, and their integrated support for decentralized declassification. The impatient reader may wish to examine Figures 1 and 2 to see the full design.

4.1 Process labels

Each process P has two labels, a *send label* P_S and a *receive label* P_R. A process's current access restrictions are stored in its send label, while the receive label holds the maximum restrictions it is willing and allowed to accept from others. The core message access check is

$$P_S \leq Q_R, \tag{1}$$

meaning P cannot send a message to Q unless P's send label is less than or equal to Q's receive label. If this check succeeds and a message is delivered, information flows from P to Q, so we *contaminate* Q with P's restrictions:

$$Q_S \leftarrow \max(Q_S, P_S).$$

These two operations are the core of any information flow system, and of many previous OS label designs [8, 18, 23].

In Asbestos, a label is a function from handles to *levels*, which are members of the ordered set $\{\star, \mathbf{0}, \mathbf{1}, \mathbf{2}, \mathbf{3}\}$ (where $\star < \mathbf{0} < \cdots < \mathbf{3}$). We write labels using set notation, such as $\{h_1\,\mathbf{0}, h_2\,\mathbf{1}, \mathbf{2}\}$. The default level, which appears without a handle at the end of the list, applies to all handles not mentioned explicitly; it is omitted when obvious from context.

To compare two labels, we compare each of their components:

$$L_1 \leq L_2 \text{ iff } L_1(h) \leq L_2(h) \text{ for all } h.$$

The least-upper-bound and greatest-lower-bound operators, max and min, are defined similarly; see Figure 1.

For example, consider processes P and Q with

$$P_S = P_R = \{h\,\mathbf{0}, \mathbf{1}\},$$
$$Q_S = Q_R = \{h\,\mathbf{3}, \mathbf{1}\}.$$

P can send to Q, since $\mathbf{0} < \mathbf{3}$; but Q cannot send to P. Low levels like $\star$ are more permissive than high values when they appear in send labels, but more restrictive when they appear in receive labels. In general, making the system more permissive should require special privilege, described further in Section 4.4.

4.2 Asymmetry

Asbestos send and receive labels default to different values. The default send label is $\{\mathbf{1}\}$, while the default receive label is $\{\mathbf{2}\}$. Furthermore, these defaults lie in the middle of the label ordering: $\mathbf{0}$ and $\star$ are less than either, and $\mathbf{3}$ is greater.

These asymmetric, intermediate defaults support more flexible isolation than more conventional designs. For example, imagine we want to prevent a process P from sending messages to a process Q, using some handle h. Asbestos supports this in three distinct ways:

	A	B	C
P_S	$\{h\,\mathbf{3}, \mathbf{1}\}$	$\{\mathbf{1}\}$	$\{h\,\mathbf{2}, \mathbf{1}\}$
Q_R	$\{\mathbf{2}\}$	$\{h\,\mathbf{0}, \mathbf{2}\}$	$\{h\,\mathbf{1}, \mathbf{2}\}$

In column A, we set $P_S(h)$ to $\mathbf{3}$. Although this prevents communication with Q as intended, P cannot send to any other process either, except for those with receive label $\{h\,\mathbf{3}\}$. Such a label change requires special privilege, since it makes the system more permissive. In column B, we instead set $Q_R(h)$ to $\mathbf{0}$, restricting the processes that can send to Q rather than those to which P can send. This time, normal processes cannot send to Q.

These isolation mechanisms, which are those available in most mandatory access control systems, limit P's ability to communicate with Q by limiting either P or Q's ability to communicate with anyone. However, Asbestos label asymmetry lets us instead express a policy involving *both* P and Q, allowing far more flexible communication while still guaranteeing restricted information flow. In column C, we set $P_S(h)$ to $\mathbf{2}$ and $Q_R(h)$ to $\mathbf{1}$. P clearly cannot send a message to Q. It can communicate with most other processes, but as it does so,

P, Q	Processes	$\star$, $\mathbf{0}$, $\mathbf{1}$, $\mathbf{2}$, $\mathbf{3}$	Label levels, in increasing order
h, *dest*	Handles	L, C, D, V, E	Labels (functions from handles to levels)

P_S	Process P's send label	$L_1 \leq L_2$	Label comparison: true if $\forall h,\ L_1(h) \leq L_2(h)$
P_R	Process P's receive label	$\max(L_1, L_2)$	Least-upper-bound label: $\{h\,k \mid k = \max(L_1(h), L_2(h))\}$
h_R	Handle h's receive label	$\min(L_1, L_2)$	Greatest-lower-bound label: $\{h\,k \mid k = \min(L_1(h), L_2(h))\}$
ctl_P	Process P's control handle	$\text{owned}(L)$	Owned-handles label: $\{h\star \mid L(h) = \star\} \cup \{h\,\mathbf{3} \mid L(h) \neq \star\}$

FIGURE 1: Notation.

it will contaminate those processes with the send label $\{h\,\mathbf{2}\}$. Thus, a process X cannot send a message to Q if it has ever communicated with P, directly or indirectly. A process that does not want to risk contamination can simply set its receive label to $\{\mathbf{1}\}$; since this is more restrictive than the default, it requires no special privilege.

4.3 Effective labels

Asbestos gives applications some control over the access control check used when sending a message. In particular, applications can further restrict access relative to the basic send and receive labels. Since the sender chooses which labels to apply, these *effective labels* can implement discretionary policies.

Senders provide a *contamination label* C_S and a *verification label* V along with every message. These labels are used to construct effective send and receive labels E_S and E_R, as follows:

$$E_S = \max(P_S, C_S), \quad E_R = \min(Q_R, V).$$

The kernel then allows delivery only if $E_S \leq E_R$. This implies that $P_S \leq Q_R$, so Equation (1) would also allow delivery. E_S, rather than P_S, is used to contaminate the receiver's send label. Finally, the kernel reports the contents of V to the receiver when delivering the message.

For example, imagine a trusted multi-user file server. Two handles u_I and u_C might be allocated for each user ID u. A process that speaks for user u would have send label $\{u_I\,\mathbf{0}\}$ and receive label $\{u_C\,\mathbf{3}\}$. When sending data intended exclusively for user u, the file server sets C_S to $\{u_C\,\mathbf{3}\}$. Only processes with receive label $\{u_C\,\mathbf{3}\}$ can receive the resulting message. After receiving, their send labels rise to $\{u_C\,\mathbf{3}\}$, and they lose the ability to talk to non-u processes (which have receive label $\{u_C\,\mathbf{2}\}$). When *receiving* data from user u, the file server conversely checks the V label, approving the message only if $V(u_I) \leq \mathbf{0}$. Processes speaking for u will set $V = \{u_I\,\mathbf{0}\}$ on each message.

Processes can also control the messages they are allowed to receive by manipulating their *handle labels*. Each handle has its own label, which can be arbitrarily set by the handle's controlling process. The handle label additionally restricts the process-wide receive label for messages sent to that handle. When P sends a message to Q over handle *dest*, the effective receive label is actually

$$E_R = \min(Q_R, dest_R, V).$$

send$(dest, C_S, D_S, V, D_R, \text{data})$
  Let Q be *dest*'s controlling process
  Let $E_S = \max(P_S, C_S)$
  Let $Q_{newR} = \max(Q_R, D_R)$
  Let $E_R = \min(Q_{newR}, dest_R, V)$
  Let $Q_{own} = \text{owned}(Q_S)$
 Requirements:
  (1) $E_S \leq E_R$
  (2) $D_R \leq dest_R$
  (3) If $D_S(h) < \mathbf{3}$, then $P_S(h) = \star$
  (4) If $D_R(h) > \star$, then $P_S(h) = \star$
 Effects:
  Grant D_S, contaminate with E_S, then restore owned handles
  $Q_S \leftarrow \max(\min(Q_S, D_S), E_S)$
  $Q_S \leftarrow \min(Q_S, Q_{own})$
  $Q_R \leftarrow Q_{newR}$

new_handle(L)
  Let h be an unused handle
 Effects:
  $h_R \leftarrow L$
  $h_R(h) \leftarrow \mathbf{0}$
  $P_S(h) \leftarrow \star$
  Return h

set_handle_label$(dest, L)$
 Requirement:
  dest was created by P
 Effect:
  $h_R \leftarrow L$

FIGURE 2: Some Asbestos label operations. P is the calling process.

By controlling the distribution of these differently-privileged handles, processes can implement a wide range of policies.

4.4 Ownership and decontamination

Finally, the special $\star$ level lets processes distribute handle access and declassify information in a decentralized way. A process with $P_S(h) = \star$ is said to *own* handle h and has two privileges with respect to it. First, P cannot be contaminated by other processes with respect to h; P's send label will stay at $\{h\star\}$ even if it receives a message with effective send label $\{h\,\mathbf{3}\}$. Second, P can *decontaminate* other processes—lower their send labels or raise their receive labels—with respect to h. Senders provide two *decontamination labels* D_S and D_R with every message, in addition to the contamination and ver-

ification labels C_S and V. The D_S label attempts to make the receiver's send label more permissive by lowering some of its levels; conversely, the D_R label attempts to make the receiver's *receive* label more permissive by raising some of its levels. As we mentioned, both of these operations are privileged, but that privilege is simply modeled by $P_S(h) = \star$. The kernel ensures that whenever $D_S(h) < \mathbf{3}$ or $D_R(h) > \star$, the process has $P_S(h) = \star$. Then, assuming the message is delivered, the kernel sets

$$Q_S \leftarrow \max(\min(Q_S, D_S), E_S) \text{ and}$$
$$Q_R \leftarrow \max(Q_R, D_R),$$

to actually accomplish the decontamination.

Decontamination of receive labels is particularly sensitive, since it can allow concurrent or later contamination. A process may not want to allow itself to become arbitrarily contaminated. Therefore, the kernel checks the D_R label in an additional way: it rejects the message unless $D_R \leq dest_R$. Since processes have full control over their handle labels, they have full control over how much contamination they are willing to accept.

A process can decontaminate other processes with respect to any handles it creates, since the **new_handle** system call sets $P_S(h) = \star$ for each created handle h. This allows the process to freely distribute its handles to others. The **new_handle** call also sets $h_R(h) = \mathbf{0}$. Since this is less than the default level **1**, only processes that have been explicitly declassified with respect to h can send messages to h. The kernel never reuses handles, so the result of a **new_handle** call has never been declassified before; a process can be sure that it can control the set of processes that can send to h by controlling h's distribution. If the process wants to accept messages from anyone, it can simply reset the handle label using a **set_handle_label** system call. Finally, a **setlabel** system call lets a process give up ownership of a handle; **setlabel** can contaminate even handles at level $\star$.

Ownership solves, in a decentralized way, problems that require global trust in most label systems: information declassification and label initialization. For instance, the ownership rules let handles implement capability semantics. When a process P creates a handle h, the kernel sets $P_S(h) \leftarrow \star$ and $h_R(h) \leftarrow \mathbf{0}$. Since the kernel never reuses handles, every other process Q must have $Q_S(h) \geq \mathbf{1}$ and can't send messages to h. To allow sending to h, P sets $Q_S(h)$ to **0** or $\star$ using a decontamination label; $\star$ lets Q grant the capability to others, while a process with value **0** can send a message to h but cannot transfer that right, either directly or by acting as a proxy. Of course, if P doesn't want h to act as a capability, it can reset h_R or contaminate its $P_S(h)$ label; capability semantics are possible but not required.

4.5 Examples

The values stored in the send and receive labels of a process govern the process' ability to communicate with other entities. Using the unique features of the Asbestos label system, such as asymmetry and the split process label design we are able to implement a wide range of security schemes. We now present, as a sample of the possibilities, two standard security schemes that can be implemented using Asbestos labels.

4.5.1 Dynamic security and process isolation

First, we consider two slightly different ways in which a process P can isolate a process Q, restricting the processes to which Q can send and receive messages.

In one case, which we call *dynamic security*, Q can send messages only to P but can receive messages from anyone. Assume that P owns Q's control handle and thus can change its receive label. As mentioned earlier, P can restrict Q by increasing a portion of its send label to a higher level than that in any other process's receive label—except, of course, for P. To accomplish these requirements, P generates a new handle, j, which has at most a default value of **2** in all processes' receive labels; then sets $Q_R(j) = Q_S(j) = \mathbf{3}$ and $P_R(j) = \mathbf{3}$ resulting in the following label setup:

Labels	P	Q	Others
Send	$j\star$	$j\,\mathbf{3}$	$j\,\mathbf{1}$
Receive	$j\,\mathbf{3}$	$j\,\mathbf{3}$	$j\,\mathbf{2}$

Q can still receive messages from anyone. Messages sent from Q to processes other than P will not go through since j's level in all other processes' receive labels has the default value of **2**, and thus *send* requirement 1 (Figure 2) does not stand.

In another policy, *process isolation*, P wants to completely isolate Q so it can send and receive messages only from P. This starts out exactly as in the dynamic security policy, but P goes further by restricting Q's ability to receive. It generates a new handle, k, which has at least the default value of **1** in all processes' send labels and sets $Q_R(k) = Q_S(k) = \mathbf{0}$ resulting in the following setup:

Labels	P	Q	Others
Send	$j\star, k\star$	$j\,\mathbf{3}, k\,\mathbf{0}$	$j\,\mathbf{1}, k\,\mathbf{1}$
Receive	$j\,\mathbf{3}, k\,\mathbf{2}$	$j\,\mathbf{3}, k\,\mathbf{0}$	$j\,\mathbf{2}, k\,\mathbf{2}$

Q is now able to receive messages only from processes whose send label contains k at a level less than or equal to 0. Since the default level for the send label is **1**, only P can now send messages to Q.

4.5.2 Multi-level security

Most MLS systems are static, with predefined secrecy levels. Asbestos labels can implement a dynamic policy, including specific security levels and arbitrary sub-levels. This implementation is also virtualizable; there may be several groups of processes participating in different MLS schemes at the same time.

Each process participating in a MLS space has a notion of current and maximum security level. In this scheme "maximum level" is analogous to a security clearance. Process P

cannot send a message to process Q if P's current security level is above Q's maximum security level. In this example, we present a scheme with secret and top secret processes (represented by handles s and t, respectively), as well as unclassified processes.

Parts of the TCB[1] (such as the identity server and the multi-level filesystem) possess $s\star$ and $t\star$. A process M is in charge of this MLS space and is part of the TCB. Process P authenticates itself to M who is able to set P's labels to correspond to the security level it should be able to use.

All processes start out with current security level set to "unclassified," and thus security handles in the send label set to their default level, **1**. Once a process S's security level is set to "secret," perhaps because it accesses a secret file from the multi-level filesystem, S can no longer send messages to unclassified processes. Since this is a restriction of S's ability to send, some handle in S_S must increase in level. This is accomplished by including a contaminate argument C on the send message from the filesystem to S, setting $S_S(s) = \mathbf{3}$, resulting in the following labels:

Labels	**U**	**S**	**T**
Send	$s\,\mathbf{1}$	$s\,\mathbf{3}$	$s\,\mathbf{1}$
Receive	$s\,\mathbf{2}$	$s\,\mathbf{3}$	$s\,\mathbf{3}$

S is now able to send messages only to processes that have had their receive label explicitly relaxed with respect to s, i.e., processes with security level "secret" (S and T in the example above).

When T (whose security level is "top secret") receives a message from S (after S has accessed secret data), the $\{s\,\mathbf{3}\}$ label will contaminate T, restricting it to sending messages to "secret" and "top secret" processes only If T receives "top secret" information, $T_S(t)$ will be increased as well leading to the following label state:

Labels	**U**	**S**	**T**
Send	$s\,\mathbf{1}, t\,\mathbf{1}$	$s\,\mathbf{3}, t\,\mathbf{1}$	$s\,\mathbf{3}, t\,\mathbf{3}$
Receive	$s\,\mathbf{2}, t\,\mathbf{2}$	$s\,\mathbf{3}, t\,\mathbf{2}$	$s\,\mathbf{3}, t\,\mathbf{3}$

4.5.3 Discussion

These examples of security mechanisms are varied in their semantics, and yet the label scheme we have presented is able to implement them both. Many other security policies may be implemented with labeling. Our label scheme is simple yet powerful. The most advantageous part is that the correctness of the policy implementation relies only on simple label properties.

4.6 Implementation

In user space, a label is represented as an array of handle values plus a default level. A 64-bit number can represent a label entry: the upper 61 bits are the handle value, the lower 3 bits encode its level in that label.

[1]Trusted computing base

In kernel space, labels are implemented similarly, but the array uses 32-bit pointers to vnodes; since those pointers are 8-byte aligned, the lower 3 bits are again available for the level. With reference counting and "copy-on-write" updates, multiple entities can share the same label, curtailing memory use. To minimize allocation, small vnode arrays are stored in the same structure as the label header. Although label operations, such as comparison and max, are quite common in Asbestos, we have not yet optimized our implementation. Nevertheless, Section 8.1 presents the results of some micro-benchmarks.

5 LABEL SAVE/RESTORE

Asbestos labels as described so far apply at the coarse granularity of entire processes. This makes sense for an operating system: the OS can control information flow between processes using hardware protection mechanisms, such as virtual memory page protection, but process internals are a black box. Unfortunately, such coarse-grained labels would make it impossible to run untrusted processes that speak for different users at different times—an important application category that includes most services. If a process can speak for more than one user, the OS must essentially trust it to keep user data internally isolated. Alternately, language-level flow control mechanisms can enforce access control policies at a much finer granularity, although communication with the outside world—i.e., other processes—can be awkward to model.

This section presents Asbestos's *label save/restore* functionality, which implements a middle ground. With label save/restore, a single untrusted process can encompass any number of independent *label spaces*. Information flow between these spaces is strictly protected by the operating system, at the granularity of memory pages.

5.1 Design

Consider our example of a Web server running multiple independent services, each of which is handling multiple concurrent connections for multiple users. We would like to prevent services from modifying one another's data and keep each user's data isolated from other users. Given Asbestos's flexible labels, this would be relatively easy, if we had one process per service–user pair. Unfortunately, OSes typically have poor support for such a large number of processes. The goal of label save/restore is to enforce the same security guarantees as this fully forked model, but at a lower cost.

Our mechanism design was motivated by the non-blocking event-driven architecture that underlies many of today's fast servers [17, 29, 35, 37]. Although some of these servers present the user with a thread-like API, all of them can look like single-process event-driven servers from the OS's perspective. These servers are built around a simple scheduling loop; roughly:

```
while (1) {
```

```
    get_next_event();
    process_event();
}
```

Note that at the end of each iteration, this connection loop has no stack or register state that depends on the event processed during that iteration. This is intentional: servers are designed to handle multiple concurrent and independent users whose events are interleaved in arbitrary order; state from one event will likely not be useful for the next. Thus, as long as the system preserves any changes to heap connection state, it would be safe to fork at the beginning of each iteration, and exit at the end. Assume, then, that we fork once per iteration. The "next event"—which, in Asbestos, is a message and its associated label contamination and decontamination—is delivered into the forked copy. Thus, the original process's label never changes, giving us exactly the label properties we want. Each message is delivered into a process with pristine labels, and "subprocesses" belonging to different users do not contaminate one another.

Of course, heap connection state must be maintained. A subprocess should have write access to its heap pages. Any data belonging to other users should be inaccessible; in fact, other subprocesses' changes to VM state should be invisible. The shared stack can be visible—it is by definition untainted—but any changes should be copy-on-write (so the subprocess cannot leak information back to the base process). A fully-forked model, with one independent service per user, would of course preserve changes to shared state; we are giving up some flexibility by exiting each subprocess at the end of the event loop. However, this flexibility isn't critical for most event-driven services since, again, they aim to perform independently for different users.

Finally, there is the matter of how a subprocess should be defined. In Asbestos, each subprocess is defined by its *send label*. To mark subprocess memory, we additionally attach a label to each physical page in the system. A subprocess's persistent heap space then consists of those pages with the same label as the subprocess.

This subprocess-like functionality is implemented with three system calls, **vm_save**, **vm_restore**, and **page_taint**, and VM page table manipulations. The **vm_save** call saves the current process's page table and register state and waits for a message to arrive. When a message arrives, it figures out which subprocess to run by calculating the labels induced by message delivery. It then updates the process's page tables to give appropriate access to memory (read/write, copy-on-write, or none) based on page labels. When the process calls **vm_restore**, the subprocess page table is thrown away and the process jumps back to the **vm_save** call. Finally, subprocess memory is allocated by **page_taint**, which marks existing pages with a given label.

To illustrate, consider a Web server with one subprocess per user. The main event loop would look up connection state information for each incoming message, process that request, then restore to a pristine state to wait for the next message. In Asbestos:

```
1  vm_save(&msg);
2  if (new_state_msg(msg))
3     new_connection_state(msg);
4  else
5     cxn = lookup_cxn_state(msg);
6     process_user_message(msg, cxn);
7  vm_restore();
```

The server saves its state before accepting any messages, allowing it to return to that pristine state. Notifications of new users (line 2) are delivered in uncontaminated messages. This allows the main subprocess to allocate and taint a portion of virtual memory for the new user. Without this untainted message, the user subprocess would not be able to communicate its existence to the main subprocess, and there would be no way to prevent different users' connection state areas from colliding. However, the uncontaminated process only knows which users are active, and where it has allocated memory for them; it cannot access actual user data. Other messages (lines 5–6) are contaminated by the user's label, and thus start the user subprocess. All memory changes are transient, except for changes to memory contaminated with the user's handle. Data for other users is inaccessible, and private user information cannot be exported to the untainted process (the relevant pages are copy-on-write), so any messages that contain user data will be contaminated with the user's label.

5.2 Implementation

The **vm_save** call first saves copies of the current page table, register state, and labels. **vm_save** takes two arguments, a message queue identifier (that is, a handle) and a message pointer; it also saves these arguments. Then **vm_save** delivers a message by: selecting a message from the specified message queue; processing its labels, creating the send label corresponding to the relevant subprocess; setting up the page table appropriately for that subprocess; delivering the message into the pointer specified by **vm_save**'s caller; and returning. The **vm_restore** call throws away the current page table, restores register state and labels to the saved versions, and delivers a message, using the same code for delivery as **vm_save**.

The interesting step in this process is setting up the page table. Figure 3 shows pseudocode for the relevant function, **fork_pages**.

The **label_cmp** function determines whether or not a page belongs to a particular subprocess. If, for some handle *h*, the page's label at *h* is greater than the subprocess's label at *h*, then the page belongs to a different user and is marked inaccessible. Otherwise, if, for some handle *h*, the page's label at *h* is *less* than the subprocess's label at *h*, then the page is less contaminated than the subprocess and should be marked copy-on-write. Otherwise, the labels are essentially equal, and the subprocess should share a (possibly writable) copy with the main process. Note that this lets one subprocess view

any changes made in strictly-less-contaminated subprocesses. For instance, consider a save/restore process with saved label $L_0 = \{h\,\mathbf{0}, \mathbf{1}\}$ and subprocess labels $L_1 = \{h\,\mathbf{1}, \mathbf{1}\}$ and $L_2 = \{h\,\mathbf{2}, \mathbf{1}\}$. The most contaminated subprocess L_2 will be able to see changes made by both L_0 and L_1, although its own changes are invisible to them. Although a deviation from the forked-subprocess model, this is safe enough: since $L_0 \leq L_1 \leq L_2 \leq P_R$, the L_0 and L_1 subprocesses could communicate with L_2 by sending it an appropriately contaminated message.

Handle ownership (that is, handles with $P_S(h) = \star$) adds further complications. A process that owns a handle is privileged with respect to that handle; as part of that privilege, the process may grant the handle to others. In label save/restore, we interpret this to mean that a subprocess that gets ownership of a handle may grant that ownership to later instances of itself. That is, we want handles owned by a subprocess to still be owned when the subprocess wakes up again. Unfortunately, page labels won't list those owned handles, and a naive comparison of send labels might cause the system to fork a new copy-on-write subprocess each time ownership changes (since the labels are not exactly the same). Thus, **label_cmp** ignores owned handles in its comparisons, and we remember and restore each subprocess's full set of owned handles.

The saved page directory and page tables are treated as having page labels equal to the saved label. Thus, if a delivered message doesn't change the process's send label, the "subprocess" works directly on the saved page table; any changes it makes will persist across the next **vm_restore**. However, any subprocess with a different send label will work on a copy-on-write version of the page tables. It may allocate, free, and taint memory however it likes, but its changes will disappear when it calls **vm_restore**, with one safe exception. Any changes the tainted subprocess makes to portions of virtual memory *not mapped in the saved page table* may be made persistent (as long as the allocated pages are tagged with the subprocess's send label). This is because untainted subprocesses cannot distinguish between unmapped memory and memory tagged with a tainted label. This leads to significant memory savings in practice. On learning that a new user session is about to start (which is uncontaminated information), the subprocess allocates a virtual memory region for that user; but it need not allocate physical memory for that region. Instead, the user's own subprocess will allocate physical memory for it as needed.

Our current implementation of these system calls is at best naive. We copy page tables more frequently than required, walk the entire page table on every save/restore (rather than just the portions that might be relevant to a given subprocess), and flush the TLB with abandon. None of these problems are fundamental.

label_cmp(A, B)
  If $\exists h$ with $A(h) > B(h) > \star$,
    Return $+1$
  Else if $\exists h$ with $\star < A(h) < B(h)$,
    Return -1
  Else return 0

fork_pages(L)
  If **label_cmp**$(L_{\text{save}}, L) = 0$,
    Share page tables
    // But continue to fix page table permissions
  Else
    Share page tables copy-on-write
  For each page mapping P,
    (1) If **label_cmp**$(P.\text{label}, L) > 0$,
      // Page belongs to a different subprocess
      Mark P inaccessible
    (2) Else if P is read-only,
      Share P read-only
    (3) Else if **label_cmp**$(P.\text{label}, L) < 0$,
      // Page is less contaminated than L
      Share P copy-on-write
    (4) Else if **label_cmp**$(P.\text{label}, L) = 0$,
      // Page belongs to L's subprocess;
      // our modifications should be saved.
      // But the saved version might be COW.
      // If so, copy it now, so we can
      // distinguish this case from (3).
      If P is copy-on-write, then copy it
      Share P

FIGURE 3: **fork_pages**.

5.3 Discussion

Reasoning about Asbestos labels, and the more clearly safe, yet analogous, model of subprocess forking, led us to modify the behavior of other system calls, and our applications, to preserve safety. For instance, our original **vm_save** design did not include message delivery. It simply restored the virtual memory state, registers, and labels to their original values; the process would continue on uncontaminated until it executed a normal **recv**. This opened a storage channel: the uncontaminated process could discover both how many contaminated messages were delivered, and their order relative to any uncontaminated messages. We therefore integrated **vm_restore** with the single operation that might or might not contaminate the process, namely **recv**, and disallowed the direct use of **recv** within a save/restore block. For the final version, we will further investigate this and other system calls—including, for example, a call to **vm_save** within an existing save/restore block—to determine the maximal safe set of operations.

6 USER-LEVEL DESIGN

Most of the applications developed for Asbestos are services, where careful consideration of information flow and access control was necessary. In particular, we ensured that if a front-end service was compromised, the effects would be isolated.

6.1 Fundamental services

Identity server To maintain the notion of users, an identity service, *idd*, maintains a list of all users. For each user, *idd* maintains the username, password, user id, and a *taint handle* and *grant handle* for that user. A user possessing the grant handle speaks for that user. A user possessing the taint handle knows some private data about that user.

"System" processes (the file server and network server) can obtain all grant and taint handles at $\star$ from *idd*. This allows the file server or network server to handle contaminated data without becoming contaminated.

Any process can request information about a user. The process must specify the username and password for the user it is interested in; if they match, *idd* replies with the user id, the grant handle at $\star$, and the taint handle at **3**.

File server It is necessary to have some form of persistent storage in Asbestos, for programs, data files, etc. To do so, the file service, *asfs*, maintains an on-disk filesystem. The file system uses the taint handles from *idd* to provide multi-user access; when a process accesses filesystem data, it becomes tainted with that user's taint handle.

6.2 Network server

All access to the network in Asbestos is through one process, *netd*, which is responsible for interfacing with the TCP stack, managing network devices, and creating connections for other processes. As such, it has a privileged role and must properly apply restrictions to connections it creates.

An application can send a message to *netd* requesting a connection to a remote host or to listen for incoming connections. In either case, *netd* performs the requested operation and grants a handle representing the new socket to the application at $\star$.

Once a process has a handle to an open connection, it may perform `READ` and `WRITE` operations to transfer data, `CONTROL` operations to close the connection or change the low-water mark, and `SELECT` operations to determine available buffer space. On a listening socket, a process may perform `READ` operations to accept incoming connections and `CONTROL` operations to close the socket.

In order to apply labeling to network connections, *netd* optionally maintains a *taint* for each connection. A process may tell *netd* to add a taint (a handle) to a connection. Later, when *netd* sends a message in response to an operation on a connection, it contaminates the recipient with the taint at **3**.

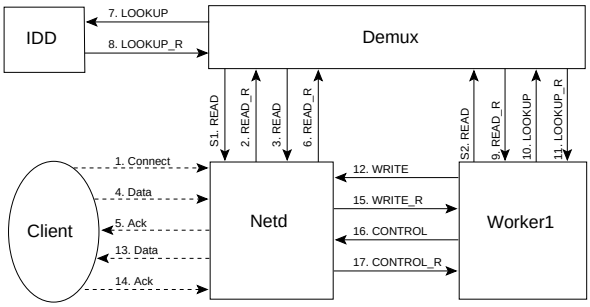

FIGURE 4: The sequence of messages when processing a Web request.

6.3 Web server

The Asbestos Web server is an alternate implementation of the OKWS design [17]. In the original OKWS design, one *demultiplexer* accepted incoming connections and parsed the headers to determine what service was being requested. The connection was then handed off to a *worker* devoted to providing that service. The goal was to isolate services, so that one compromised service could not affect others.

The Asbestos implementation of OKWS isolates services in different workers, but also enforces user isolation, in order to prevent one compromised service from leaking information about other users. Rather than using a separate process for each user, OKWS uses **vm_save** and **vm_restore** to provide full isolation of one user's data from others.

6.3.1 The launcher

OKWS first starts *launcher*, which starts the separate OKWS components and ensures proper communication privileges for each. The launcher creates *N worker verify handles* (where *N* is the number of services), one for each *worker*. These are used to verify that a worker process is valid.

The *launcher* starts *demux* first, which grants its own handle (the *demux handle*) to the launcher. After starting *demux*, *launcher* grants *demux* each of the worker verify handles.

Next, *launcher* starts *N worker*s, granting each *worker* the demux handle. This allows each *worker* to contact *demux* to announce that it is ready to service a request. The *launcher* also grants each *worker* its worker verify handle.

6.3.2 The demux

After sending the demux handle to *launcher*, *demux* waits for each *worker* to contact it. When a *worker* contacts *demux*, it passes its worker verify handle, to prove to *demux* that it is a valid *worker*. After verifying *worker W*, *demux* creates an new handle and sends it to *W*. The *worker* later uses this handle to communicate with *demux*.

This sequence of events for handling connections is shown in Figure 4. The *demux* contacts *netd* and opens a listen socket for incoming connections (message S1). When a connection arrives (messages 1–2), *demux* reads enough of the HTTP headers to determine what user *U* is making the

request, and what worker *W* that user is requesting (messages 3–6). Once it has done so, it contacts *idd* to obtain the taint handle for that user, sending the username and password (message 7), receiving *U*'s taint handle (message 8) in response. At this point, the *demux* is ready to hand the connection off to *W*; however, it must ensure that as soon as *W* reads from *U*'s socket, it becomes tainted with *U*'s label.

6.3.3 The workers

An OKWS system will typically have many workers running, each implementing a logically distinct Web service. Each service sends a `READ` message (message S2) when it first starts up, expressing interest in incoming requests. When the *demux* is ready to hand a particular worker (call it *W*) a connection, it simply replies to this `READ` message (as in message 9). The worker then immediately can reply with another `READ` message (as in S2), since it is capable of serving overlapping clients.

The handoff shown in messages 9 through 11 requires care. Each worker maintains server-side state for each active user it is communicating with. This state includes receive buffers, send buffers, and perhaps cached session information that might persist over multiple HTTP sessions. In our implementation, each worker *W* sets aside a 64-page region for each user *U* that becomes active, and it allocates pages there lazily, as the user requires them. Moreover, it taints this entire region with *U*'s taint handle, so that it might later write to it when it has *U*'s taint handle in its send label.

If the *demux* were to deliver *U*'s connection tainted with *U* immediately in message 9, the worker would be at a loss. It would have to set aside a region for *U*'s state, but it could not write to any persistent, *non-tainted* memory to indicate that it had done so. OKWS on Asbestos instead uses a two-phase handoff protocol. In message 9, the *demux* informs *W* that it is about to deliver a connection for user *U*, but *does not taint W* as it does so. *W* then consults a table *T* (implemented as a quadratic hash table), either finding a previously allocated region for *U*, or allocating a new one. Note that *T* resides in *uncontaminated* memory. When *W* writes to *T* that it has allocated a region for the user *U*, it can later read this mapping without becoming contaminated.

After noting this new region assignment in *T*, *W* is ready to accept the connection and the taint, and it does so in message 10 and 11. Once it has received *U*'s connection, it enjoys a reserved, pretainted region of pages, to which it can write persistently when contaminated by *U*'s handle. The *worker* then services *U*'s connection, parsing the request, sending the reply back to the client over the connection (messages 12–15), and closing the connection (messages 16–17). Once complete, *worker W* calls **vm_restore** and waits to service another connection.

One interesting issue remains: freeing memory. When worker *W* corresponds with *U* over many HTTP requests, it can grow and shrink *U*'s region while tainted, always leaving at least one page allocated to store a map of the region. When the user *U* explicitly logs off, *W* would like to reclaim the last page, and reassign *U*'s region to other users who become active. To achieve this, *W* sends a message to the *demux*, informing it that *U*'s region should now be available to other users. The *demux* immediately sends back an acknowledgment, telling *W* to free the last page in the region. The demux then waits a random amount of time, on the order of ten minutes, and then sends a second *uncontaminated* message, telling *W* to mark *U*'s region unallocated in the table *T*.[2] The region is now available for reassignment to a different user. Note the *demux* must be careful to synchronize this last messages with potential requests from *U*, so as to avoid race conditions on *W*.

7 COVERT CHANNELS

Asbestos labels prevent processes from explicitly transmitting sensitive information to unauthorized parties. However, supposedly isolated processes can still communicate information through covert channels. Our goal is not to eliminate covert channels—an impossible task—just to make it significantly harder to leak information than on systems used as Internet servers today. While high-grade military systems are required to quantify all covert channels, for Asbestos we content ourselves with enumerating the channels.

Broadly speaking, covert channels can be categorized as either timing or storage channels. Timing channels consist of attacks in which process *A* conveys information to *B* by modulating its use of system resources in a way that observably affects *B*'s response time. For instance, *A* might flush the processor cache or cause the disk arm to be moved farther from a subsequent request. We are less concerned with timing channels, as to some extent they can be mitigated by limiting processes' ability to measure time precisely [10]. (Asbestos offers no such feature, however, and the problem admittedly becomes harder in the presence of network communication.)

Storage channels are caused by any state that can be modified by process *A* and observed by *B* when *A* is not supposed to transmit information to *B*. It was a goal to avoid storage channels that could be exploited within a single process, so that at least two cooperating processes are required to communicate information in violation of a label policy.

The Asbestos design contains two inherent storage channels: the program counter, and labels. The **vm_restore** system call affects the program counter of an untainted process by restarting a process at its save point with a lower send label. Two cooperating processes can, for instance, transmit a bit of information by the order in which they call **vm_restore**. This channel is roughly equivalent to the covert channel intentionally included by the drop-on-exec feature of IX [23].

The **send** system call potentially raises the value of the

[2] *demux*'s response represents a storage channel, which we mitigate by a long delay.

recipient's send label to an unanticipated value. This is also a storage channel, as labels can be observed through lack of communication. Consider a tainted process A attempting to communicate a bit of sensitive information to an untainted process C. An attacker might construct two untainted processes, B_0 and B_1, both of which repeatedly send heartbeat messages to C. By sending a message that contaminates process B_i, A can communicate the value i to C. Such storage channels are inherent to any system with run-time checking of dynamic labels [27].

Both of the above channels require at least two processes, which means they can be mitigated by restricting access to the fork device. This illustrates one advantage of the **vm_save**/**vm_restore** page labeling approach, compared to a more traditional one-label-per-process architecture. Page labeling reduces concurrency, thereby also reducing the number of send labels and program counters available as storage channels at any given time.

Other Asbestos kernel data structures have been carefully designed to avoid exploitable storage channels. Handles are generated by incrementing a 61-bit counter, which is a storage channel. However, since the kernel encrypts the counter value with a 61-bit block cipher to produce handles, the user-visible sequence of handles is unpredictable and thus cannot convey information. The VM system prevents page tables and page labels from being used as storage channels by ensuring changes made while tainted are not visible with a lower send label.

The current implementation still has several other storage channels we intend to close or limit, but we believe these can be mitigated without affecting the claims of the paper. For example, Asbestos does not yet deal gracefully with certain forms of resource exhaustion. Also, when a process sends a message to an invalid handle, or to another process whose label prevents it from replying, we intend for **send** always to return the error `E_MAYBE`. The implementation does not yet do this uniformly.

8 EVALUATION

In this section, we present micro-benchmarks of the Asbestos kernel and an end-to-end analysis of the performance of the services running on top of it. We identify areas where performance becomes a bottleneck and further work is necessary.

8.1 Micro-benchmarks

Most Asbestos operations involve label operations. In particular, all messages exchanged in the system require label checks and potential label modifications, and page labeling makes extensive use of label operations during save/restore sessions. To evaluate the performance of Asbestos we need to quantify the cost of label operations. We conducted a set of micro-benchmarks that exercise seven basic label operations that are considered most common and potentially costly.

Operation	size = 50	size = 100	size = 200
label_create	1510	1530	1531
label_min	4198	5511	8078
label_max	5521	5561	7644
label_leq	586	1028	2021
label_cmp	540	1040	2130
label_add	179	199	223
label_add (COW)	188	223	246

FIGURE 5: Label operations average cost measured in cycles.

The functions evaluated are the following:

label_create : create a label of given size,
label_min and **label_max** : generate a label by applying the *min* and *max* operators shown in Figure 1 to its arguments[3],
label_add : add a handle to a label,
label_add (COW) : add a handle to a label triggering the copy-on-write code,
label_leq : "less than or equal" operator (see Figure 1),
label_cmp : label comparison operation returning "less than," "equal," and "greater than" values.

All measurements where taken on a PC equipped with a 2.8GHz Pentium4 processor with 1MB L2 cache and 1GB of RAM, running Asbestos. All operations were measured using labels with 50, 100 and 200 handles. The numbers presented are averages for 3 runs of 100 iterations each. Figure 5 shows the results of all seven micro-benchmarks.

label_create performance is dominated by the (constant) memory allocation cost. The average cycles for **label_min** and **label_max** are dominated by a particular test case whose cost was an order of magnitude higher than that of all other test cases. That happened mainly due to memory allocation and copying involved in that test. All other investigated scenarios showed that min and max operations scale linearly with the size of the labels, as did **label_leq** and **label_cmp**, whose behavior was very stable with no odd results. Both **label_add** operations' results showed that the size of the label does not affect results significantly, mainly because the cost is dominated by the time spent sorting the array of label components. Triggering "*copy-on-write*" has a minimal performance hit since our current label implementation uses a label allocation "arena" (implemented as a free-list) that speeds up label duplication.

Micro-benchmark results revealed certain points that could be optimized, but in general we were able to show that, using our untuned label implementation, the average cost of operations is reasonable and scales well with the size of the labels involved.

8.2 End-to-End Measurements

As a proof of concept, we implemented and measured a version of OKWS running on Asbestos. We did not expect

[3]Note that these operations alter their first operand, e.g. **label_min**(a, b) is equivalent to $a =$ **label_min**(a, b)

Module	Percent Execute Time
Kernel – misc	2.70
Kernel – memory mgmt	6.34
Kernel – vm-restore	28.26
Kernel – IPC	12.53
Kernel – Network	18.47
Console	6.47
User – Network	16.85
User – okws-demux	5.60
User – okws-worker	2.77

FIGURE 6: Execution time of various modules while OKWS is under heavy load.

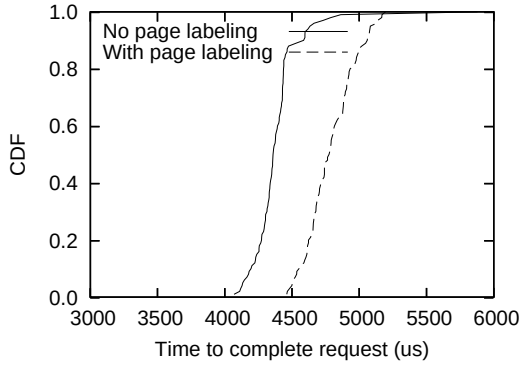

FIGURE 7: CDF of request time as seen from the client.

our system's latencies and throughput to be competitive with highly optimized operating systems or Web servers, but we did expect reasonable baseline performance numbers that we could improve upon as development continues. Performance is discussed in Section 8.2.1. More to the point, we did expect OKWS on Asbestos to be reasonably efficient in terms of memory utilization, since an additional page of memory is potentially all that is required to allocate an additional protection domain. An exploration of memory usage in OKWS can be found in Section 8.2.2.

8.2.1 Web Server Speed

As shown in Figure 6, performance is largely limited by time spent in the kernel and on networking. Over a third of the total CPU time is spent on networking.

Our baseline measurements were all taken on a 10Mb Ethernet network with a local Linux client generating requests. The server is an AMD Athlon 1400 using 64MB of memory. Clearly support for faster Ethernet drivers and more memory is critical for gaining more competitive performance numbers.

Due to bugs in our TCP/IP implementation, concurrent connections lead to timeouts and did not offer a performance gain. Figure 7 shows the overhead of page labeling. Our server is able to serve over 210 connections per second. With page labeling disabled, it is possible to serve over 240 connections per second. Page labeling currently adds about 500 μs of CPU time to each connection, primarily in **vm_restore**. A modern web server should be able to handle thousands of connections per second, and the time in **vm_restore** alone currently limits Asbestos/OKWS to 2000 connections per second. As our system matures, it will be critical to optimize **vm_restore**.

8.2.2 Memory Usage

In Section 5, we argued the merits of page labeling over the more traditional fork-accept designs. Our measurements of the Asbestos prototype's memory utilization lend credence to this claim. A minimal OKWS worker process requires at least 20 pages of physical memory (each 4KB), before it has even served an HTTP request: one page for the page directory, about four for page tables, one for a user stack, one for a user exception stack, and others for the BSS and user heap. Other pages are artifacts of our kernel and application implementation, though an operating system that uses fewer than five pages per process is difficult to imagine.

The reasonable conclusion to draw is that to support n users in the fork-accept model, OKWS would require approximately $20n$ memory pages at a minimum, a cost which will become prohibitive as n grows large. By contrast, page labeling for simple Web services achieves the lower bound that we proposed earlier: one memory page per external client served.

To experimentally verify this claim, we configured a test client to simulate 100 Web requests on behalf n different users, as n grew from 1 to 25. For each run, we captured the maximum number of pages active at any one given time. For instance, at $n = 1$, we measured 1146 total pages in use as the Web server launched and a maximum of 1264 pages in use as the 100 serialized requests were made, all on behalf of the same user. At $n = 2$, the maximum number of pages in use increases to 1265. These patterns increases roughly linearly until $n = 25$, at which point the maximum number of pages in use is 1288. Although we can improve upon absolute memory usage, the overall trend is encouraging: for some constant C, OKWS on Asbestos should support n concurrently active users with only $n + C$ pages.

9 CONCLUSION

Asbestos is an operating system that makes nondiscretionary access control mechanisms available to unprivileged users, giving them fine-grained, end-to-end control over the dissemination of information. Asbestos provides protection through a new labeling scheme, which, unlike schemes in previous operating systems, allows data to be sanitized (or "untainted") by individual users within categories they control. The categories, called handles, use the same names as communication endpoints, making them a kind of generalization of capabilities. Like capabilities, processes can dynamically generate new handles, handle ownership is aggregated by process (allowing explicit enumeration of privileges), and processes specify temporarily label restrictions on sent messages to avoid the unintentional use of privilege.

The Asbestos virtual memory system allows labels to be applied at the granularity of individual pages, so that one can

control the flow of information even within one process. A prototype web server handles labeled data in such a way that even software bugs cannot cause one user to receive another's private data. The system requires only one page of memory per active user, and exhibits a tolerable slowdown of only 12% for the vastly increased security of fine-grained information flow control.

10 ACKNOWLEDGEMENTS

Michelle Osborne

Mike Mammarella, Chris Frost (adlr? leiz?)

lwip guy

REFERENCES

[1] Viktors Berstis. Security and protection of data in the IBM system/38. In *Proceedings of the 7th Symposium on Computer Architecture*, pages 245–252, May 1980.

[2] William Earl Boebert. On the inability of an unmodified capability machine to enforce the *-property. In *Proceedings of the 7th DoD/NBS Computer Security Conference*, pages 291–293, September 1984.

[3] M. Branstad, Homayoon Tajalli, Frank Mayer, and David Dalva. Access mediation in a message passing kernel. In *Proceedings of the 1989 IEEE Symposium on Security and Privacy*, May 1989.

[4] D.R. Cheriton. The V distributed system. *Comm. ACM*, 31(3):314–333, 1988.

[5] Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, 1977.

[6] Department of Defense. *Trusted Computer System Evaluation Criteria (Orange Book)*, dod 5200.28-std edition, December 1985.

[7] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole. Exokernel: An operating system architecture for application-level resource management. In *Proc. 15th Symposium on Operating Systems Principles*, pages 251–266, Copper Mountain, Colorado, December 1995.

[8] Timothy Fraser. LOMAC: Low water-mark integrity protection for COTS environments. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, pages 230–245, Oakland, CA, May 2000.

[9] R. Goldberg. Architecture of virtual machines. In *1973 NCC AFIPS Conf. Proc.*, volume 42, pages 309–318, 1973.

[10] Wei-Ming Hu. Reducing timing channels with fuzzy time. In *Proceedings of the 1991 IEEE Symposium on Security and Privacy*, pages 8–20, Oakland, CA, May 1991.

[11] Trent Jaeger, Atul Prakash, Jochen Liedtke, and Nayeem Islam. Flexible control of downloaded executable content. *ACM Transactions on Information and System Security*, 2(2):177–228, 1999.

[12] Paul A. Karger. Limiting the damage potential of discretionary trojan horses. In *Proceedings of the 1987 IEEE Symposium on Security and Privacy*, pages 32–37, Oakland, CA, April 1987.

[13] Paul A. Karger and Andrew J. Herbert. An augmented capability architecture to support lattice security and traceability of access. In *Proceedings of the 1984 IEEE Symposium on Security and Privacy*, pages 2–12, Oakland, CA, April–May 1984.

[14] Paul A. Karger, Mary Ellen Zurko, Douglas W. Bonin, Andrew H. Mason, and Clifford E. Kahn. A VMM security kernel for the VAX architecture. In *Proceedings of the 1990 IEEE Symposium on Security and Privacy*, pages 2–19, Oakland, CA, May 1990.

[15] Key Logic. *The KeyKOS/KeySAFE System Design*, sec009-01 edition, March 1989. `http://www.agorics.com/Library/KeyKos/keysafe/Keysafe.html`.

[16] Samuel T. King and Peter M. Chen. Operating system support for virtual machines. In *Proceedings of the 2003 Annual USENIX Technical Conference*, June 2003.

[17] Maxwell Krohn. Building secure high-performance web services with OKWS. In *Proceedings of the 2004 USENIX*, Boston, MA, June 2004. USENIX.

[18] Carl E. Landwehr. Formal models for computer security. *Computing Surveys*, 13(3):247–278, September 1981.

[19] Robert Lemos. Payroll site closes on security worries, Feb 2005. `http://news.com.com/2102-1029_3-5587859.html`.

[20] Jochen Liedtke. On microkernel construction. In *In Proceedings of the Fifteenth ACM Symposium on Operating System Principles*, Copper Mountain Resort, CO, December 1995.

[21] Peter Loscocco and Stephen Smalley. Integrating flexible support for security policies into the Linux operating system. In *Proceedings of the 2001 USENIX*, pages 29–40. USENIX, June 2001. FREENIX track.

[22] Catherine Jensen McCollum, Judith R. Messing, and LouAnna Notargiacomo. Beyond the pale of MAC and DAC – defining new forms of access control. In *Proceedings of the 1990 IEEE Symposium on Security and Privacy*, pages 190–200, Oakland, CA, May 1990.

[23] M. Douglas McIlroy and James A. Reeds. Multilevel security in the UNIX tradition. *Software—Practice and Experience*, 22(8):673–694, 1992.

[24] Mark S. Miller, Ka-Ping Yee, and Jonathan Shapiro. Capability myths demolished. Technical Report SRL2003-02, Johns Hopkins University Systems Research Laboratory, 2003. `http://www.erights.org/elib/capability/duals/`.

[25] James G. Mitchell, Jonathan Gibbons, Graham Hamilton, Peter B. Kessler, Yousef Y. A. Khalidi, Panos Kougiouris, Peter Madany, Michael N. Nelson, Michael L. Powell, and Sanjay R. Radia. An overview of the Spring system. In *COMPCON*, pages 122–131, 1994.

[26] Andrew C. Myers and Barbara Liskov. A decentralized model for information flow control. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 129–142, Saint-Malo, France, October 1997. ACM.

[27] Andrew C. Myers and Barbara Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Computer Systems*, 9(4):410–442, October 2000.

[28] News10. Hacker accesses thousands of personal data files at CSU Chico, March 2005. `http://www.news10.net/storyfull1.asp?id=9784`.

[29] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. Flash: An efficient and portable Web server. In *Proceedings of the 1999 USENIX*, pages 199–212. USENIX, June 1999.

[30] Richard F. Rashid and George G. Robertson. Accent: A communication oriented network operating system kernel. In *Proceedings of the A CM Symposium on Operating System Principles*, December 1981.

[31] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Guillemont M. Gien, F. Herrmann, C. Kaiser, P. Leonard, S. Langlois, and W. Neuhauser. Chorus distributed operating systems. *Computing Systems*, 1:305–379, oct 1988.

[32] Jonathan S. Shapiro, Jonathan Smith, and David J. Farber. EROS: a fast capability system. In *Proc. Symposium on Operating Systems Principles*, pages 170–185, 1999.

[33] Andrew S. Tanenbaum, Robbert van Renesse, Hans van Staveren, Gregory J. Sharp, Sape J. Mullender, Jack Jansen, and Guido van Rossum. Experiences with the Amoeba distributed operating system. *Communications of the ACM*, 33(12):46–63, 1990.

[34] VMware. VMware and the National Security Agency team to build advanced secure computer systems, January 2001. `http://www.vmware.com/pdf/TechTrendNotes.pdf`.

[35] Rob von Behren, Jeremy Condit, Feng Zhou, George C. Necula, and Eric Brewer. Capriccio: Scalable threads for Internet services. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 268–281, Bolton Landing, NY, October 2003. ACM.

[36] Robert Watson, Wayne Morrison, Chris Vance, and Brian Feldman. The TrustedBSD MAC framework: Extensible kernel access control for FreeBSD 5.0. In *USENIX Annual Technical Conference 2003*, pages 285–296, San Antonio, TX, June 2003.

[37] Matt Welsh, David Culler, and Eric Brewer. SEDA: An architecture for well-conditioned, scalable Internet services. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 230–243, Chateau Lake Louise, Banff, Canada, October 2001. ACM.

[38] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. Scale and performance in the Denali isolation kernel. In *Proceedings of the 2002 Symposium on Operating Systems Design and Implementation (OSDI)*, December 2002.